Application Development using Compositional Performance Analysis

Adam Rifkin
Advisor: K. Mani Chandy

Department of Computer Science
California Institute of Technology
Pasadena, California 91125

Written February 29, 1996
Revised July 7, 1999

In partial Fulfillment of the Requirements
for the degree of Master of Science

Acknowledgements

There are so many people to thank for their help in this work: Berna Massingill for her help in all aspects of this project; Greg Davis for his help in developing the mesh-spectral archetype implementation; Anita Mareno, John Langford, and Lena Petrovic for their help in developing the applications used for our experiments; Argonne National Laboratory and Intel Corporation for providing access to suitable computing facilities; and Hewlett-Packard, Microsoft, and IBM for their support and resources. This research was supported in part by the NSF under CRPC grant CCR-9120008.

Special thanks go to Mani Chandy, who has nurtured me in his role as an advisor, a leader, and a friend.

Thanks go to the professors and researchers at Caltech who have inspired me: Yaser Abu-Mostafa, Jim Arvo, Al Barr, Peter Bossaerts, Colin Camerer, Joel Franklin, Mary Hall, Carl Kesselman, Alain Martin, Chuck Seitz, Peter Schröder, Eric Van de Velde, and Jan L.A. van de Snepscheut.

Thanks go to the friends at Caltech who have assisted me and supported me: Diane Goodfellow, Roman Ginis, Robert Harley, Peter Hofstee, Rohit Khare, Joseph Kiniry, Svetlana Kryukova, Rustan Leino, Rajit Manohar, Berna Massingill, Paul Sivilotti, Eve Schooler, John Thornley, and Dan Zimmerman.

And, thanks go to my parents William and Linda, brother Brian, and sister Jennifer, for always believing in me, and to my best friend and wife, Michelle, for being the light in my life, in sickness and in health.

Abstract

A *parallel programming archetype* [Cha94, CMMM95] is an abstraction that captures the common features of a class of problems with a similar computational structure and combines them with a parallelization strategy to produce a pattern of dataflow and communication. Such abstractions are useful in application development, both as a conceptual framework and as a basis for tools and techniques.

The efficiency of a parallel program can depend a great deal on how its data and tasks are decomposed and distributed. This thesis describes a simple performance evaluation methodology that includes an analytic model for predicting the performance of parallel and distributed computations developed for multicomputer machines and networked personal computers. This analytic model can be supplemented by a simulation infrastructure for application writers to use when developing parallel programs using archetypes.

These performance evaluation tools were developed with the following restricted goal in mind: We require accuracy of the analytic model and simulation infrastructure only to the extent that they suggest directions for the programmer to make the appropriate optimizations. This restricted goal sacrifices some accuracy, but makes the tools simpler and easier to use.

A programmer can use these tools to design programs with decomposition and distribution specialized to a given machine configuration. By instantiating a few architecture-based parameters, the model can be employed in the performance analysis of data-parallel applications, guiding process generation, communication, and mapping decisions.

The model is language-independent and machine-independent; it can be applied to help programmers make decisions about performance-affecting parameters as programs are ported across architectures and languages. Furthermore, the model incorporates both platform-specific and application-specific aspects, and it allows programmers to experiment with tradeoffs better than either strictly simulation-based or purely theoretical models. In addition, the model was designed to be simple.

In summary, this thesis outlines a simple method for benchmarking a parallel communication library and for using the results to model the performance of applications developed with that communication library. We use *compositional performance analysis* — decomposing a parallel program into its modular parts and analyzing their respective performances — to gain perspective on the performance of the whole program. This model is useful for predicting parallel program execution times for different types of program archetypes (e.g., mesh and mesh-spectral), using communication libraries built with different message-passing schemes (e.g., Fortran M and Fortran with MPI) running on different architectures (e.g., IBM SP2 and a network of Pentium personal computers).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sequential algorithm development can be performed in a computer-independent fashion, because the fundamental sequential computer model drafted by Von Neumann is fixed and universally accepted, and its accompanying computational complexity model is similarly well-defined and studied. Although real computers deviate from the Von Neumann model somewhat, high-level programming languages and operating systems successfully hide the differences between the actual computer and the model; hence, most programmers need only be aware of the model.

Even novice computer programmers can use computational complexity models to determine that, generally, an $O(n \log n)$ algorithm will demonstrate better performance than an $O(n^2)$ algorithm. We note that there is sufficient accuracy in the *Big Oh* notation to guide a programmer's decision in choosing the more efficient algorithm of the two. However, users who want the best-possible performance from one particular computer need a detailed knowledge of its underlying hardware.

The situation for parallel and distributed programming is fairly similar; most programs written for machines with more than one sequential computing node require some computer architecture-dependent adjustments to achieve reasonable performance. However, a simple, easy-to-use, generally predictive performance model for such machines (analogous to computational complexity for sequential machines) would be useful for the stepwise refinement of parallel and distributed programs en route to optimized code that attains better performance.

The goal of this thesis is to provide a predictive performance model for parallel and distributed programs, with the following restricted goal:

Requirement 1 *We require only enough accuracy in our performance model as is needed to suggest guidelines for a programmer in refining code.*

As with the sequential case, users who want the best-possible performance from one particular computer need a detailed knowledge of its underlying hardware.

In this chapter, we provide the motivation for having such a predictive model (Section 1.1), concisely specify the problem we wish to solve (Section 1.2), summarize our performance model solution (Section 1.3), describe our experiments and results (Section 1.4), and outline the remainder of this thesis (Section 1.5).

1.1 Motivation

Completely modeling the performance of sequential programs is difficult because of features of the underlying hardware such as paging, caching, and prefetching. Completely modeling parallel and distributed programs is even more difficult because of added behaviors such as communication latencies, the bandwidths of communication between computing nodes, context switching costs between processes on the same computing node, and discrepancies in the powers of different computing nodes. However, knowing some of these behaviors — in terms of execution time "costs," relative to each other — would be useful to a parallel or distributed programmer developing an application in a machine-independent fashion, later porting the application to a specific computer.

For example, in the IBM SP1, context switching costs are expensive relative to computing costs, so setting up multiple processes on a single SP1 processor can incur many costly context switches. A programmer would benefit from a predictive performance model that captures this cost; with this knowledge, the programmer could optimize a program ported to the SP1 from another machine by modifying the program so that it provides only a single process per node.

Another example is the cost of communication. With distributed networks and parallel multicomputers, communication is slow relative to computation, in terms of both *latency* (startup cost of messaging) and *throughput* (additional cost per unit message). Specifically, for networks of workstations connected through shared channel communication protocols like Ethernet and FDDI, communication is particularly expensive, because concurrent communications over the network are serialized. On the other hand, specialized machines such as the IBM SP2, provide fast interconnect networks that allow communications in parallel. A programmer would benefit from a predictive performance model that encapsulates the communication costs of an underlying architecture; with this knowledge, the programmer could optimize a program ported to a network

of workstations connected by Ethernet as easily as a program ported to the IBM SP2, with respect to the communication infrastructure.

For the behaviors in these two examples — context switch costs and communication costs — we can imagine a predictive performance model that is very accurate but too complex to use. We choose ease-of-use as a secondary goal:

Requirement 2 *We require our performance modeling tools to be simple enough that parallel and distributed programmers will want to use them.*

Specifically, we would like to encapsulate parallel program behaviors with a performance model that employs closed-form equation analyses or discrete-event simulations. With either case, the methodology is fixed for the programmer, so simply by plugging in the proper parameters based on the problem and implementation, that programmer receives helpful tips for refining the program.

Indeed, we could imagine a "toolbox" of "tools" (equations and/or simulations) for each of a variety of classes of application programs, and the programmer could choose the proper tools to use based on the application being written. Whenever a new tool is developed, its can be added to the toolbox:

Requirement 3 *We require our performance modeling tools to encapsulate the analysis for a set of applications, so that they can be reused and composed where needed.*

Tools that allow previous performance modeling work to be reused also adhere to *Requirement 2*, since the programmer need not develop new equations and/or simulations from scratch every time. Furthermore, *composable tools* are useful because they allow **compositional performance analysis** — enabling a programmer to decompose the performance analysis of a large system into the potentially easier performance analysis of several smaller systems.

As a final motivation, we consider why we seek a predictive performance model that consists of analysis as well as simulation. The reason is simple: we believe that this gives developers the flexibility to choose the proper tool for runtime comparisons in any given application development situation:

Requirement 4 *We require our performance modeling tools to allow comparison among different runtime executions of a program, to guide choices in computation, communication, synchronization, task decomposition, and data partitioning.*

Having presented the requirements for our performance model, we can now provide an informal problem statement.

1.2 Problem Statement

Our goal is to develop a performance model that helps speed up the time required to speed up a class of parallel applications. This model should have the following features, based on the requirements:

- **Accurate** enough to guide optimizations, as per *Requirement 1.*
- **Simple** enough for programmers to use, as per *Requirement 2.*
- **Reusable** in the sense that previous modeling work can be used again, as per *Requirement 3.*
- **Composable** enough to allow for component-wise performance modeling, also as per *Requirement 3.*
- **Analytic** enough to provide ready comparisons between given executions, as per *Requirement 4.*

Parallel program archetypes offer a reasonable testbed for the development and evaluation of such a model. With archetypes, a programmer can apply general problem solving techniques to customize an application for a specific machine with specific parallel issues in mind. Then, using the performance model, the programmer can improve program performance using systematic simulations and/or closed-form analytic equations.

In summary, we seek a reasonable parallel performance toolbox that provides analytic and simulation tools with the ability to model real algorithms running on real machines. Our model should be sufficiently flexible enough to handle a range of applications, and sufficiently extensible to encompass applications not considered by the current scope of archetypes.

1.3 Summary of Our Solution

We define a *multicomputer* to consist of a collection of computing nodes connected by a communication network. A node can be a sequential processing element with a local memory, a parallel multiprocessor with a memory shared among the node's processors, or a parallel multicomputer with distributed memories local to each of the node's processors. In a homogeneous multicomputer, each node is identical; in a heterogeneous multicomputer, a variety of different nodes can be interconnected. All nodes of the multicomputer are simultaneously active, and each node runs one or more sequential processes. The only available means of information exchange between processes requires the explicit cooperation between those processes by sending and receiving messages. Note that a network of workstations (NOW) and a traditional multicomputer (e.g., the IBM SP2) share this multicomputer architecture. Further, it is our hypothesis that

we can specify, with a small collection of parameters, the relative performances of multicomputers.

We can now restate the problem we are solving as having program portability considerations: we seek an analytic (and simulation-based) performance model that facilitates the comparison of a program being run on a NOW versus the program being run on a multicomputer. The model should encapsulate some of the performance tradeoffs when developing, performance considerations when porting, and stepwise refinements of parallel & distributed programs during design. Further, we are willing to accept a *modest* model, only requiring as much accuracy as necessary as to guide program decisions.

Application development for a multicomputer can be done on an algorithmic level independent of the details of the specific architecture, in the same way that application development for a sequential computer can be done on an algorithmic level independent of the details of that specific architecture. Specifically, *archetypes* [Cha94, CMMM95, Mas98, MC96] provide a methodology and code libraries to aid in the development of correct parallel programs by exploiting common computation and communication structures. When using an archetype to write a parallel application, a programmer often needs to predict what effect modifications to an algorithm will have on the performance of that algorithm, implemented in a specific language on a given machine.

This thesis addresses the issues involved when predicting the performance of multicomputer applications developed using archetypes, providing an underlying analytic model that uses application source code and essential machine parameters. The model consists of two parts:

- *ADAPT* (for "Application Development using Analytic Performance Tuning") supplies an architecture-independent methodology for benchmarking and subsequent performance evaluation using closed-form equations.

- *ADEPT* (for "Application Development using Experimental Performance Tuning") supplies an architecture-independent methodology for benchmarking and subsequent performance evaluation using a program simulation suite.

In terms of software tools to support performance evaluation, both the ADAPT and ADEPT models have been developed into a software program for any given application suite.

Having specified the ADAPT and ADEPT models, we develop from them the *MCM* ("Multicomputer Model"), and illustrate its use in the performance evaluation of applications developed for multicomputers and networks of personal computers using the mesh and mesh-spectral archetypes. MCM furnishes a methodology that is useful at many stages of the application development process. This methodology is briefly stated in figure 1.1, and will be described in detail in Chapter 2.

Since the ADAPT and ADEPT models form a hybrid that combines the tech-

1. *Develop* the program with computation, communication, context switching, and synchronization constructs in mind.
2. *Decompose* the program into its essential runtime-intensive routines.
3. *Benchmark* these routines on the given machine.
4. *Analyze* the program in terms of those routines, yielding closed-form equations, and/or
5. *Simulate* the program in terms of those routines, yielding stochastic potential runs.
6. *Refine* the program based on the analysis and/or simulations.
7. *Repeat* steps 1-6 as needed.

Figure 1.1: A brief overview of the MCM methodology.

niques of benchmarking and performance evaluation through both analysis and simulation, and they contain a methodology specifically developed to complement the use of archetypes, their techniques allow the same pattern scavenging and reuse that make archetypes such useful application-development tools for constructing a range of applications including ozone concentration modeling, computational fluid dynamics, and electromagnetic computations. Since purely analytic performance models do not make use of actual application source code, and since simulation-based prediction tools do not allow system architects to experiment with different tradeoffs in system parameters, our hybrid approach affords a more accurate model than would a pure approach in its use with archetypes.

1.4 Summary of Results

We briefly outline the key contributions of this thesis, by identifying the fruits of this work — general principles and performance results — in an objective, scientific way, giving both pros and cons.

The work described here offers the following:

1. An analytic model for comparing application performances, complete with a notation for specifying and a methodology for using.

2. A simulation model for comparing application performances, complete with a notation for specifying and a methodology for using.

3. A multicomputer performance model that combines the analytic and simulation models as a tool to guide design decisions during application development.

4. A methodology for using the multicomputer performance tool to model algorithm classes, as demonstrated through the co-development of archetype solutions.

5. Several experiments using the multicomputer performance model, showcasing its strengths and weaknesses.

The pros of our contributions are manyfold. We present a new way of thinking about parallel computation modeling that incorporates many bootstrapped techniques used in other research, casing them together in a general, employable methodology. Since our an hybrid approach to performance modeling for multicomputers leverages off archetypes, we can adopt certain conventions and assumptions that make the model simple but useful. Although we maintain a synergy with archetypes, we are not reliant on them; we briefly discuss later how our performance model can be used independently of archetypes.

The cons of our model reside in its oversimplification of some of the parallel program design decisions: since we do not seek a model that perfectly models the intended program implementation (instead seeking a model that adequately models the implementation to guide program design decisions), the model is not an excellent predictor of program performance. However, we demonstrate that it still has much utility in helping parallel programmers during the performance refinement process.

1.5 Organization of this Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the model, related work, and our experimental methods. Chapter 3 and Chapter 4 describe applications based on the mesh archetype and the mesh-spectral archetype, for use in our performance experiments. Chapter 5 discusses the experiments, and Chapter 6 presents conclusions.

Chapter 2

Performance Model

Archetype-based performance models exploit commonalities in programs to simplify the process of performance analysis. The performance model in this thesis is based on the ideas of extrapolation from observations, asymptotic analysis, scalability analysis, execution profiles, and data fitting, as investigated by Foster [Fos95, Chapter 3].

2.1 Related Work

The general scheduling problem is: Given a set T of n tasks, a partial order $<$ on T, weight W_i, $1 \leq i \leq n$, m processors, and a time limit k, does there exist a total function h from T to $\{ 0, 1, ..., k-1 \}$ such that

- If $i < j$, then $h(i) + W_i \leq h(j)$
- For each i in T, $h(i) + W_i \leq k$
- For each t, $0 \leq t < k$, there are at most m values of i for which $h(i) \leq t < h(i) + W_i$

This problem was proven to be NP-complete by Karp in 1972. El-Rewini and Lewis catalog a list of variations on the scheduling problem that are also NP-Complete; furthermore, this complexity does not improve when communication costs are considered [ERL98]. For example, Papadimitriou and Yannakakis proved that the problem of optimally scheduling unit-time task graphs with communication on an unlimited number of processors is NP-complete when the communication between any pair of processors is the same and greater than or equal to one [PY79]. As a result, developers often look to heuristics (for example, see Mao et al.'s heuristics for on-line algorithms for the single machine scheduling problem [MKR95]) or simulation methods (for example, see Zahorjan

et al.'s stochastic modeling of the effect of scheduling disciplines on spin overhead in shared memory parallel systems [ZLE91]).

To provide solutions to real-world scheduling problems, restrictions on the parallel program and the machine representations can be relaxed. For example, since the problem of choosing the data partitioning and distribution to achieve the optimal performance is NP-complete, we are more interested in *user-guided* performance evaluation tools for the refinement of parallel applications than in automatic performance prediction (for example, Fahringer's work with P^3T [Fah96a]). Since our model is likely to be used to supplement a programmer's efforts to develop applications using archetypes, it differs from efforts to do performance measurement for compiler optimization (as Clement and Quinn do with C^* on multicomputers [CQ93]), and it differs from efforts to estimate performance statically to automate the load balancing of useful work within a program (as with Fahringer's application of P^3T [Fah96b]).

Many models have been developed to provide a simple but accurate model of parallel computation to aid in parallel algorithm design. Like the frequently used Parallel Random Access Machine (PRAM) model [FW78, KR90] and Bulk Synchronous Parallel (BSP) model [Val90, GV94], our model decomposes programs into fairly large blocks; our model, however, incorporates the idea of archetypes, gaining ease of use at the expense of greater generality. Like the LogP model [CKP+93], our model captures both communication bandwidth and communication latency through parameters; however, whereas LogP models communication in terms of the four parameters latency (L), overhead (o), bandwidth (g), and number of processors (P), our model's parameters for any given application are based on the archetypes being employed in the development of that application.

Our techniques fit in well with other methodologies for dealing with applications developed for particular architectures (for example, Brinch Hansen's model for programming multicomputer applications [BH93a]). Archetypes frequently represent well-researched patterns or abstractions; for example, the mesh archetype [Mas96] builds on Brinch Hansen's work on parallel cellular automata in the context of multicomputers [BH93b]. In that paper, the computational complexity of parallel cellular automata is derived and shown to be a sufficiently accurate estimator of the performance of a Laplace's equation solver; in this thesis, we provide an alternative technique of performance estimation using a combination of benchmarking and analysis that is especially suited to applications developed using archetypes. The mesh-spectral archetype [DM96] extends the parallel cellular automata model, providing row and column operations in addition to grid operations.

2.2 Methodology

A great deal of work has been done both on methods of exploiting design patterns in program development (for example, [Col89] and [GHJV95]) and on methods of solving problems on concurrent processors (for example, [FJL+88] and [BBC+93]). *Archetypes* [Cha94, CMMM95] were developed as design patterns with the single restricted goal of modeling one kind of pattern that is relevant in parallel programming: the pattern of the parallel computation and communication structure.

Methods of exploiting design patterns in program development begin by identifying classes of problems with similar computational structures and creating abstractions that capture the commonality. Combining a problem class's computational structure with a parallelization strategy gives rise to a dataflow pattern and hence a communication structure. It is this combination of computational structure, parallelization strategy, and the implied pattern of dataflow and communication that we capture as a *parallel programming archetype* [Mas98, MC96]. For the remainder of this paper, we use the term "archetype" to refer to a parallel programming archetype.

A key question in the development of a parallel application, especially for a multicomputer or a network of computers, is the issue of data decomposition and distribution. Archetypes directly address the question of which data distributions are compatible with a problem's computational structure. In some cases more than one data distribution is compatible with the computational structure; in such cases, which one is chosen does not affect program correctness, but it may well affect program performance.

In this paper and an earlier paper [Rif93] we address the question of how the choice of data distribution affects performance and present a methodology for archetype-based application development, including a phase in which a correct but not necessarily efficient application is refined to improve performance by using an archetype-based performance model.

Our methodology for designing, developing, implementing, and refining an application follows the workflow schedule illustrated in Figure 2.1. This workflow consists of six major phases: Analysis, Application, Benchmarking, Performance Model, Simulation, and Refinement, each of which we describe in turn.

Analysis. In the Analysis phase, an appropriate archetype is chosen to help with application development. If more than one archetype could be used, our performance models can suggest which one is more efficient, as is done in Section 5.1. This phase usually starts with a problem description and/or a sequential program to be parallelized; it ends when the appropriate archetype has been chosen. Two phases of the methodology can then be worked on concurrently: development of the application (path (a) in Figure 2.1), and benchmarking of computation and communication routines (path (b) in Figure 2.1).

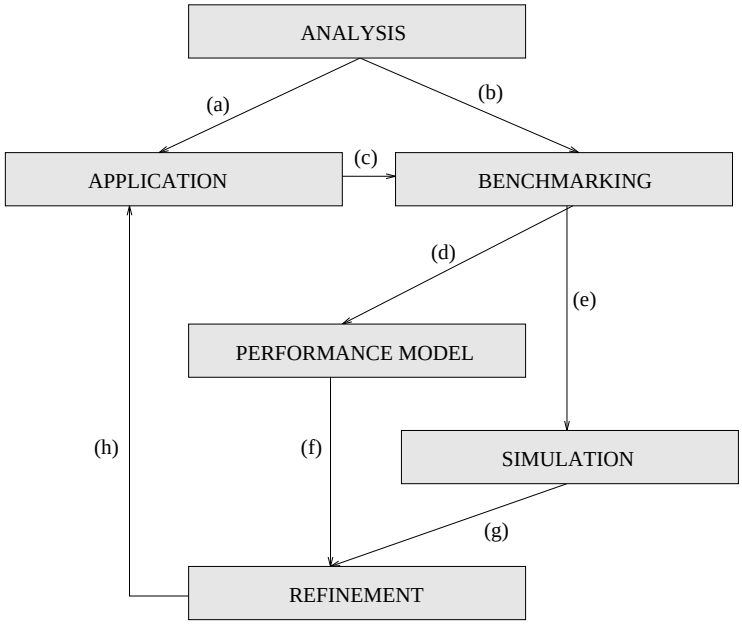


Figure 2.1: The methodology for using and analyzing the performance of archetypes follows a simple workflow.

Application. In the APPLICATION phase, the selected archetype is used to develop an application to solve the specified problem or to parallelize the given sequential program, as follows.

First, the programmer develops an initial archetype-based version of the algorithm. This initial version is structured according to the archetype's pattern and gives an indication of the concurrency to be exploited by the archetype. Essentially, it is produced by adapting the original algorithm or program to fit the archetype pattern and "filling in the blanks" of the archetype with application-specific details. An important feature of this initial archetype-based version of the algorithm is that it can be executed sequentially; if the algorithm is deterministic, it can also be debugged sequentially.

This initial version is then transformed into an equivalent version suitable for efficient execution on the target architecture. The archetype assists in this transformation, either via guidelines to be applied manually or via automated tools. Again, the transformation can optionally be broken down into a sequence of smaller stages, and in some cases intermediate stages can be executed (and debugged) sequentially. A key aspect of this transformation process is that the

transformations defined by the archetype preserve semantics and hence correctness.

The programmer then implements the efficient archetype-based version of the algorithm using a language or library suitable for the target architecture. Here again the archetype assists in this process, not only by providing suitable transformations (either manual or automated), but also by providing program skeletons and/or libraries that encapsulate details of the parallel code (for example, process creation and message-passing). A significant aspect of this step is that it is only here that the application developer must choose a particular language or library; the algorithm versions produced in the preceding steps can be expressed in any convenient notation, since the ideas are essentially language-independent.

After this implementation is tested and debugged, its performance can be evaluated and possibly improved via the remaining phases of our methodology.

Benchmarking. BENCHMARKING of archetype communication and computation routines can be performed during or independently of the APPLICATION phase. This phase collects data to be used in performance evaluation via analytic techniques and/or simulation methods.

Our performance model requires two sets of benchmarks: archetype-specific ("communication") benchmarks and application-specific ("computational") benchmarks. Both sets involve measuring execution times of the relevant routines (from the archetype library for the first set, and from the application for the second) using the target architecture, language, compiler, and library. Computational benchmarks can be done on a single processor; communication benchmarks require N processors, where N is the maximum number of processors the application might use. If the programmer wants to use analysis to help choose an appropriate granularity, communication benchmarks must be done for varying numbers of processors.

Our model uses approximations of higher-level communication routines rather than approximations of low-level measurements such as latency and bandwidth. We do this for two reasons: It is a very simple method for gathering information about the potential performance of a program, and it allows developers to reuse benchmark measurements for applications using the same archetype. Recall the restricted goal: We require accuracy of the analytic model and simulation infrastructure only to the extent that they suggest directions for the programmer to make appropriate optimizations.

Once the relevant routines are benchmarked, the programmer can use the results in the PERFORMANCE MODEL phase (path (d) in Figure 2.1) as well as in the SIMULATION phase (path (e) in Figure 2.1). Our experiments show that usually the PERFORMANCE MODEL phase is sufficient (without the SIMULATION phase) for predicting program efficiency and for doing the corresponding performance tuning.

Performance Model. The PERFORMANCE MODEL phase consists of two steps:

1. *Analysis* of the program to produce a closed-form equation involving the benchmarked quantities, and
2. *Instantiation* of the equation with the appropriate benchmarked values to give a number representing a prediction of the program's expected running time.

We call the methodology of developing the program concurrently with modeling its performance ADAPT, for Application Development using Analytic Performance Tuning.

The *Analysis* segment of ADAPT involves decomposing the given program into a number of subprograms (e.g., initialization, computational loop, and termination) whose running times can be expressed in terms of the benchmark numbers. The finer the grain of decomposition and benchmarking, the more predictive we expect the model equation to be for that program.

The basis for this decompositional approach to performance modeling is a structured induction on the statements of the program being modeled, assuming implicit barriers between any subprograms. For example, for a program S that consists of program S_1 (which may involve a number of computations and communications) composed in sequence with program S_2 (which also may involve a number of computations and communications), we assume an implicit barrier between S_1 and S_2:

$$S = S_1; S_2$$

For the base case in which S_1 and S_2 each consist of either a single computation, using one or more of the given processors, or a (possibly collective) communication operation also using one or more of the given processors, we can model the expected running time $\mathcal{R}(S)$ of program S as follows: S_2:

$$\mathcal{R}(S) = \mathcal{R}(S_1) + \mathcal{R}(S_2)$$

Since S_1 consists of a single computation or communication, we model $\mathcal{R}(S_1)$ as the maximum of the expected running times of that computation or communication on all of the processors. If $\mathcal{R}_p(S_1)$ is the expected running time of S_1 on processor p, we can model $\mathcal{R}(S_1)$ thus:

$$\mathcal{R}(S_1) \approx \max_{\forall p} \mathcal{R}_p(S_1)$$

Note that this model underestimates the running time because the maximum of the expected times is at most the expected value of the maximum running time; in practice, for the algorithms we investigated, the model provides a useful approximation. We can model the expected running time of $\mathcal{R}(S_2)$ similarly; as a result, we have:

$$\mathcal{R}(S) \approx \max_{\forall p} \mathcal{R}_p(S_1) + \max_{\forall p} \mathcal{R}_p(S_2)$$

The structured induction thus allows us to compute the expected running time of a large program from the expected running times of its components. For example, having derived a closed-form equation for $\mathcal{R}(\mathrm{S})$, we can use it to compute expected running of a program T that consists of two executions of $\mathcal{R}(\mathrm{S})$ in sequence:

$$\begin{aligned} \mathcal{R}(\mathrm{T}) &\approx \mathcal{R}(\mathrm{S}) + \mathcal{R}(\mathrm{S}) \\ &\approx 2 \times (\max_{\forall p} \mathcal{R}_p(\mathrm{S}_1) + \max_{\forall p} \mathcal{R}_p(\mathrm{S}_2)) \end{aligned}$$

Part (a) of Figure 2.2 illustrates how this equation models running time on two processors. Because the model uses implicit barriers, we expect that it will yield conservative or pessimistic performance estimates (i.e., predicted execution times possibly greater than actual execution times), as illustrated in part (b) of the figure. We can derive a similar equation for a program that consists of NSTEPS iterations of S:

$$\begin{aligned} \mathcal{R}(\mathrm{T}) &\approx (\sum_{i=1}^{\mathrm{NSTEPS}} \mathcal{R}(\mathrm{S})) \\ &\approx \mathrm{NSTEPS} \times (\max_{\forall p} \mathcal{R}_p(\mathrm{S}_1) + \max_{\forall p} \mathcal{R}_p(\mathrm{S}_2)) \end{aligned}$$

We demonstrate the use of this simple performance model in Sections 3, 4, and 5.

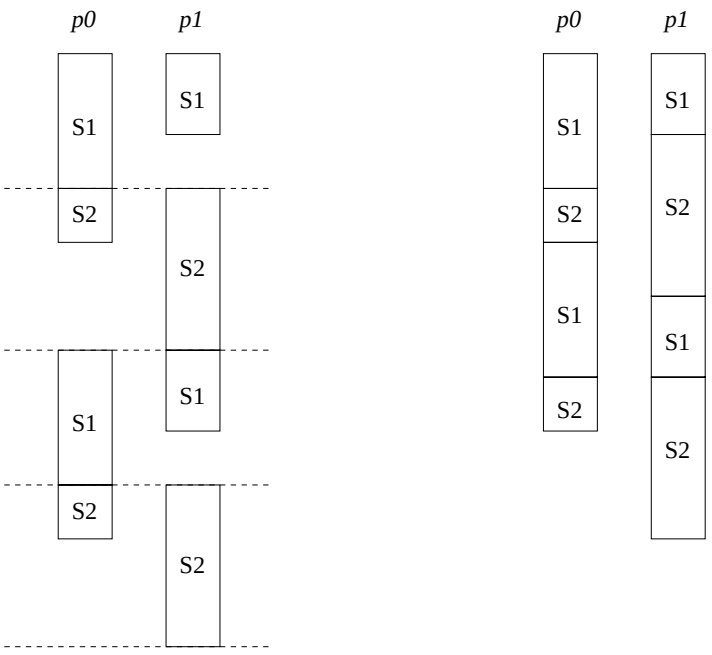
The *Instantiation* segment of ADAPT is simply the "plugging in" of benchmark numbers into the closed-form equations developed during the *Analysis* segment, either manually or via an automated tool.

The PERFORMANCE MODEL phase can (but need not) be done concurrently with the SIMULATION phase. With or without the supplementary simulations, ADAPT can guide the programmer in making decisions about data distributions and granularity during the REFINEMENT phase (path (f) in Figure 2.1).

Simulation. The SIMULATION of a program consists of writing a simulation program based on the actual application program being modeled. This phase supplements the PERFORMANCE MODEL phase, in cases in which the programmer would like to consider additional factors before making data partitioning and granularity decisions for large program runs.

We call the methodology of developing the program while using the performance model with a simulation architecture ADEPT, for Application Development using Experimental Performance Tuning. Like the performance model, these simulations can help guide a programmer in making decisions about data distribution and granularity during the REFINEMENT phase (path (g) in Figure 2.1).

Briefly, setting up a simulation works as follows. Normally, each routine being benchmarked in the BENCHMARKING phase is timed repeatedly and the



(a) Execution time predicted using implicit barriers. (b) Possible actual execution time.

Figure 2.2: Predicted versus actual execution times for program S_1; S_2; S_1; S_2 on two processors. (a) illustrates how our model predicts execution times for using implicit barriers between program units. (b) illustrates a possible actual execution scenario, if in fact barriers are not needed.

results are combined into an average execution time for the routine. If simulation is to be done, however, rather than simply averaging the results the programmer records their distribution. He or she then writes a program analogous to the application program, with the actual application program statements replaced by the generation of estimated running times based on the distributions observed during benchmarking. Rather than computing the results of the application, the simulation computes the expected running time of the application; this can be done quickly on a single processor as many times as the user desires.

Refinement. In the REFINEMENT phase, choices regarding data distribution and granularity are reconsidered based on expected running times as computed

in the PERFORMANCE MODEL phase. Using ADAPT and ADEPT, the programmer can return to the APPLICATION phase (path (h) in Figure 2.1) to improve the efficiency of the application if necessary. The performance model (and simulation strategy) may suggest a decision between archetypes (Section 5.1), architectures (Section 5.2), or libraries (Section 5.3).

2.3 Experimental Method

We use a small suite of application programs developed using two different archetypes to explore the PERFORMANCE MODEL phase of the methodology described in Section 2.2. This section describes our experimental method.

System specifications. Our experiments were conducted using various combinations of two different archetypes (mesh and mesh-spectral), two different architectures (an IBM SP2 at Argonne National Labs using a straight interconnect, and a network of 166 MHz Pentium personal computers connected by 100Mbps Ethernet), and two different languages/libraries (Fortran M and Fortran with MPI).

Fortran M [FC95] consists of a small set of extensions to Fortran for modular parallel programming. In Fortran M, tasks and channels are represented explicitly, allowing the design of structured, unstructured, and asynchronous communication patterns for task-parallel programs. In addition, Fortran M gives control over the mapping of tasks to processors. We use the TCP/IP-based implementation of Fortran M for our Fortran-M based programs.

MPI [Mes94, SOHL+96], the Message Passing Interface, is a standard, portable message-passing system that defines the syntax and semantics of a package of library routines useful to a wide range of applications written in Fortran or C. Several free, well-tested, efficient implementations of MPI exist, both for distributed memory multicomputers and networks of personal computers and workstations.

Environment. Experiments were performed on otherwise unloaded computer nodes, with one application process per node, but in an environment in which communications hardware was also supporting other users. Since execution times were consistent across multiple runs, we assume that this sharing of communications hardware did not greatly affect our results.

Measurement of execution times. We measured two kinds of execution times:

- Elapsed "total" time, measured using the Unix `time` command, includes the time required to start the processes.

- Elapsed "process" time, measured by calling the Unix `gettimeofday` system function from within each process, does not include any overhead associated with starting and ending processes.

That is, "total" time is measured from program initiation to program termination, while "process" time is measured from process initiation to process termination. Note that we do not measure elapsed "computational kernel" times, which might show more scalability for our algorithms.

Averaging. Measurements are averaged over several trials, with high and low outliers discarded. Every experiment is done at least twice to verify the consistency of the results.

Presentation of results. We provide tables containing benchmark measurements, and graphs illustrating observed running times versus those computed using the performance model. In the tables, times are rounded to the nearest integer, so very small times are shown are zero. We plot execution times rather than speedups; each plot shows the following:

- *Ideal* time is sequential execution time (on 1 processor) divided by number of processors — that is, time required for a program with ideal speedup.
- *Actual* time is observed time as measured by our experiments.
- *Expected* time is calculated using the appropriate performance model and the results of our benchmark experiments.

The full text of the experimental parallel programs, sequential programs, and benchmark programs are presented in the appendix of [RM96].

Chapter 3

Mesh Applications

In an application based on the mesh archetype [Mas96], data is based on an N-dimensional grid ($N = 1$, 2, or 3), with one or more variables per cell (grid point), and computation consists of some sequence of the following operations:

Computing, for each cell, new values for one or more variables, based on old values of variables in that cell and nearby cells (for example, neighbors or next-to-neighbors).

- (Optionally) reading in values for a grid variable.
- (Optionally) writing out values for a grid variable.
- (Optionally) computing a global reduction (for example, global maximum or global sum) over the whole grid.

Frequently the compute-new-values and reduction operations are performed repeatedly in a time-step loop. Figure 3.1 illustrates the basic operation of computing new values in terms of old values, in a two-dimensional grid.

Mesh computations are readily parallelized for distributed-memory architectures using the following approach: The N-dimensional grid is distributed over an N-dimensional grid of processes; computation of new values for grid variables is similarly distributed. A separate (optional) host process is used for reads/writes involving a whole array. Non-grid variables (for example, global constants and reduction variables) are duplicated in each process and their values are kept consistent. This approach is discussed by Massingill [Mas96], including details about how to parallelize sequential code and about how to build an application program from the archetype-provided code template and library. The library includes routines that encapsulate the necessary communication operations (host-to-grid and grid-to-host redistribution, boundary exchange, reductions, and broadcast) and a number of utility routines for index manipulation

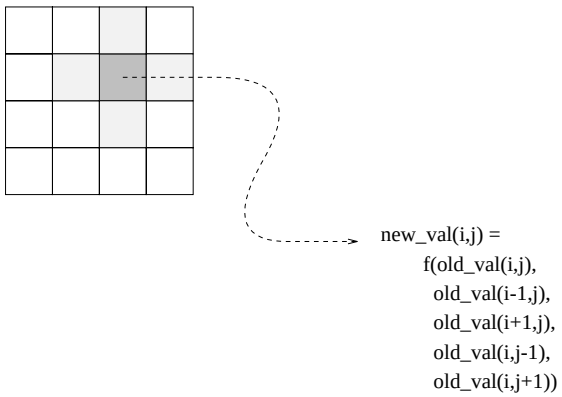


Figure 3.1: Basic mesh computation.

and other housekeeping. This archetype code (template and library) has been implemented in Fortran M, Fortran with Intel's NX Library, and Fortran with p4 [BL94]. All implementations have essentially the same application programmer interface, so applications developed using one implementation are trivially ported to another.

We note that for the two mesh archetype applications given in Sections 3.1 and 3.2, the total process count includes a host process for performing I/O, which affects performance. There is also a no-host-process version of the mesh archetype, but doing I/O with that version is somewhat more complicated, and in the applications described in this paper we chose programming simplicity over performance where such trade-offs had to be made.

3.1 Heat Diffusion

The heat diffusion application [Mas96] solves the one-dimensional heat diffusion equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

using the approximation:

$$\frac{U(x_i, t_{k+1}) - U(x_i, t_k)}{\Delta t} = \frac{U(x_{i+1}, t_k) - 2U(x_i, t_k) + U(x_{i-1}, t_k)}{\Delta x^2}$$

A sequential program for this computation is straightforward. It maintains two copies of variable U, one for the current time step (`uk`) and one for the next time step (`ukp1`). At each time step, it computes the values of `ukp1` based on the

values of `uk`. The two boundary points are handled differently; they maintain a constant value.

An equivalent parallel program using the mesh archetype is similar: Grid-based variables `uk` and `ukp1` are distributed among grid processes. Each local section is surrounded by a "ghost boundary" of width one, to be used to hold values from neighboring processes. The whole array is initialized in the host process and then copied to the grid processes. (It could also be initialized directly in the grid processes; this approach was chosen for simplicity.) At each time step, the ghost boundaries are updated (by calling the archetype's boundary-exchange routine) before they are used in the grid computation. The special handling for the (global) boundary points is provided by using an archetype library routine to determine which points in each local section are in the interior of the global array. The grid values are then copied back to the host process for printing. The code executed by the host and grid processes has the same high-level structure; both execute the time-step loop, for example. This ensures that proper synchronization is maintained. Both programs appear in the appendix of [RM96].

Benchmarking. The computational benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate process), $\mathcal{T}_{\text{init}}$ (initialize grid values), $\mathcal{T}_{\text{comp}}$ (calculate new values for all grid points), and $\mathcal{T}_{\text{output}}$ (output results). Results are given in Table 3.1. Observe that results for this benchmark are independent of the choice of archetype implementation.

Measurement	Time (msecs)
$\mathcal{T}_{\text{overhead}}$	150
$\mathcal{T}_{\text{init}}$	27
$\mathcal{T}_{\text{comp}}$	124
$\mathcal{T}_{\text{output}}$	15

Table 3.1: Results of computational benchmark for the mesh heat diffusion application, running on a single node of the IBM SP2 using Fortran. Grid size is 640,000 points. Times are in milliseconds.

The communication benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate processes), $\mathcal{T}_{\text{HtoG}}$ (redistribute data, host to grid), $\mathcal{T}_{\text{xintersect}}$ (compute appropriate local indices), $\mathcal{T}_{\text{update\_bdry}}$ (update ghost boundaries by exchanging data with neighboring processes), and $\mathcal{T}_{\text{GtoH}}$ (redistribute data, grid to host). We ran this benchmark on 1, 2, 4, 8, 16, and 32 processors (plus a "host" processor, as described earlier). Results are given in Table 3.2.

Measurement	Time (msecs) on n nodes, not including host node					
	$n=1$	$n=2$	$n=4$	$n=8$	$n=16$	$n=32$
$\mathcal{T}_{\text{overhead}}$	8650	9980	14330	16230	24480	68130
$\mathcal{T}_{\text{HtoG}}$	556	649	935	940	991	1048
$\mathcal{T}_{\text{xintersect}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{update\_bdry}}$	0	6	8	9	9	9
$\mathcal{T}_{\text{GtoH}}$	530	533	549	553	516	435

Table 3.2: Results of communication benchmark for the mesh heat diffusion application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 640,000 points. Times are in milliseconds.

The computational and communication benchmark programs appear in the appendix of [RM96].

Performance Model. We use our performance model and the program, given in the appendix of [RM96], to compute estimated running time in terms of the preceding list of benchmark values and two additional program parameters — NSTEPS (number of loop iterations) and XPROCS (number of grid processes) — as follows. First, a high-level decomposition gives us values for $\mathcal{T}_{\text{elapsed}}$ (total estimated elapsed time) and $\mathcal{T}_{\text{process}}$ (total estimated process time, excluding process-setup overhead):

$$\begin{aligned} \mathcal{T}_{\text{elapsed}} &= \mathcal{T}_{\text{process}} + \mathcal{T}_{\text{overhead}} \\ \mathcal{T}_{\text{process}} &= \mathcal{T}_{\text{startup}} + \mathcal{T}_{\text{computation}} + \mathcal{T}_{\text{finish}} \end{aligned}$$

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are as follows:

$$\begin{aligned} \mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{init}} \\ \mathcal{T}_{\text{computation}} &= \text{NSTEPS} \times \mathcal{T}_{\text{comp}} \\ \mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{output}} \end{aligned}$$

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines,

as follows:

$$\begin{aligned}
\mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{init}} + \mathcal{T}_{\text{HtoG}} + \mathcal{T}_{\text{xintersect}} \\
\mathcal{T}_{\text{computation}} &= \text{NSTEPS} \times \left(\frac{\mathcal{T}_{\text{comp}}}{\text{XPROCS}} + \mathcal{T}_{\text{update_bdry}} \right) \\
\mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{GtoH}} + \mathcal{T}_{\text{output}}
\end{aligned}$$

Experimental Results. For this experiment we used the following values of the program parameters:

$$\begin{aligned}
\text{NSTEPS} &= 2000 \\
\text{XPROCS} &= n
\end{aligned}$$

where n is the number of grid (non-"host") processors (1, 2, 4, 8, 16, or 32). Table 3.3 and Figures 3.2 and 3.3 compare predicted with observed times. For this experiment, predicted times generally agreed well with observed times, with predicted times being, as we expected, somewhat conservative. Cases in which we overestimated expected elapsed time (for example, for $n = 32$) were due to benchmarked overhead costs that turned out to be not as significant to the actual program's running time. Our model also correctly predicts the scalability of the application, validating its utility in helping programmers choose granularity.

	Time (secs) on n nodes, not including host node					
	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
Expected Elapsed Time	258	147	94	66	59	96
Actual Elapsed Time	237	130	76	54	43	48
Expected Process Time	249	137	79	50	35	27
Actual Process Time	231	123	67	39	24	19

Table 3.3: Execution times for the mesh heat diffusion application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 640,000 grid points and 2000 steps. Times are in seconds. See Figures 3.2 and 3.3 for corresponding graphs.

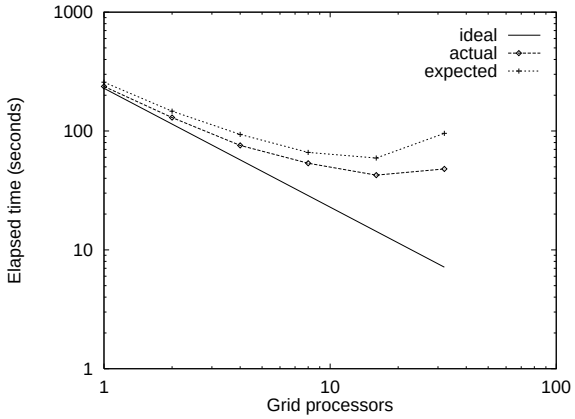


Figure 3.2: Elapsed times for the mesh heat equation application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 640,000 grid points and 2000 steps. Times are in seconds. See Table 3.3 for corresponding table.

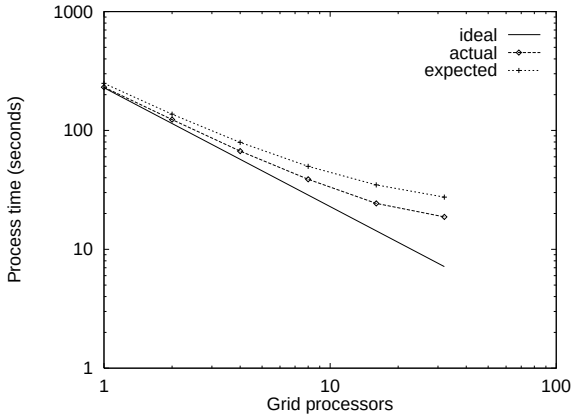


Figure 3.3: Process times for the mesh heat equation application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 640,000 grid points and 2000 steps. Times are in seconds. See Table 3.3 for corresponding table.

3.2 Poisson Solver

This application [Mas96], based on the discussion of the Poisson problem by Van de Velde [VdV94], solve the equation

$$-\frac{\partial^2 U}{\partial x^2} - \frac{\partial^2 U}{\partial y^2} = f(x,y)$$

with Dirichlet boundary condition

$$u(x,y) = g(x,y)$$

using Jacobi iteration; that is, by discretizing the problem domain and applying the following operation to all interior points until convergence is reached:

$$4u_{(i,j)}^{(k+1)} = h^2 f_{i,j} + u_{(i-1,j)}^{(k)} + u_{(i+1,j)}^{(k)} + u_{(i,j-1)}^{(k)} + u_{(i,j+1)}^{(k)}$$

Convergence is said to be reached when the maximum of

$$|u_{(i,j)}^{(k+1)} - u_{(i,j)}^{(k)}|$$

falls below a specified tolerance.

A sequential program for this computation maintains two copies of variable u, one for the current iteration (`uk`) and one for the next iteration (`ukp1`). At each iteration, it computes the values of `ukp1` based on the values of `uk`. The boundary points are handled differently; they maintain a constant value. Every `NCHECK` the maximum difference between `uk` and `uk` is computed to check for convergence. At each step, values are copied back from `ukp1` to `uk` (for the sake of simplicity, at a modest cost in performance).

An equivalent parallel program using the mesh archetype is similar: Grid-based variables `uk` and `ukp1` are distributed among grid processes. Each local section is surrounded by a "ghost boundary" of width one, to be used to hold values from neighboring processes. The whole grid is initialized in the host process and then copied to the grid processes. (It could also be initialized directly in the grid processes; this approach was chosen for simplicity.) At each time step, the ghost boundaries are updated (by calling the archetype's boundary-exchange routine) before they are used in the grid computation. The special handling for the (global) boundary points is provided by using archetype library routines to determine which points in the local section are in the interior of the global array. Computing a global maximum for the convergence test is accomplished by computing a local maximum in each grid process and then calling an archetype library routine to find the global maximum. When convergence is reached (or `MAXSTEPS` iterations have been performed), grid values are copied back to the host process for printing. The code executed by the host and grid processes has the same high-level structure; both execute the main loop, for example, including the convergence test. This ensures that proper synchronization is maintained. Both programs appear in the appendix of [RM96].

Note that this problem can also be solved using the mesh-spectral archetype, as described in Section 4.2. We compare the performances of the two implementations (mesh and mesh-spectral) in Section 5.1.

Benchmarking. The computational benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate process), $\mathcal{T}_{\text{init}}$ (initialize grid values), $\mathcal{T}_{\text{comp}}$ (calculate new values for all grid points), $\mathcal{T}_{\text{check\_converge}}$ (check for convergence), $\mathcal{T}_{\text{copy\_values}}$ (copy new values back to `uk`), and $\mathcal{T}_{\text{output}}$ (output results). Results are given in Table 3.4. Observe that results for this benchmark are independent of the choice of archetype implementation.

Measurement	Time (msecs)
$\mathcal{T}_{\text{overhead}}$	100
$\mathcal{T}_{\text{init}}$	247
$\mathcal{T}_{\text{comp}}$	430
$\mathcal{T}_{\text{check\_converge}}$	128
$\mathcal{T}_{\text{copy\_values}}$	45
$\mathcal{T}_{\text{output}}$	15

Table 3.4: Results of computational benchmark for the mesh Poisson solver application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

The communication benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate processes), $\mathcal{T}_{\text{iglobal}}$ (index manipulation), $\mathcal{T}_{\text{jglobal}}$ (index manipulation), $\mathcal{T}_{\text{GtoH}}$ (redistribute data, grid to host). $\mathcal{T}_{\text{HtoG}}$ (redistribute data, host to grid), $\mathcal{T}_{\text{merge\_real\_maxabs}}$ (merge local maxima into global maximum), $\mathcal{T}_{\text{update\_bdry}}$ (update ghost boundaries by exchanging data with neighboring processes), $\mathcal{T}_{\text{xintersect}}$ (index manipulation), and $\mathcal{T}_{\text{yintersect}}$ (index manipulation). We ran this benchmark on 1, 4, 9, 16, 25, and 36 processors (plus a "host" processor, as described earlier). Results are given in Table 3.5.

The computational and communication benchmark programs appear in the appendix of [RM96].

Performance Model. We use our performance model and the program, given in the appendix of [RM96], to compute estimated running time in terms of the preceding list of benchmark values and a few additional program parameters — NSTEPS (number of loop iterations), NCHECK (frequency of convergence checking), and XPROCS and YPROCS (dimensions of process grid, implying a total of

Measurement	Time (msecs) on n nodes, not including host node					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
$\mathcal{T}_{\text{overhead}}$	6	8	14	18	33	37
$\mathcal{T}_{\text{iglobal}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{jglobal}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{GtoH}}$	618	630	626	697	667	639
$\mathcal{T}_{\text{HtoG}}$	665	1103	1116	1211	1275	1409
$\mathcal{T}_{\text{merge\_real\_maxabs}}$	7	52	69	83	121	177
$\mathcal{T}_{\text{update\_bdry}}$	0	11	17	18	25	21
$\mathcal{T}_{\text{xintersect}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{yintersect}}$	0	0	0	0	0	0

Table 3.5: Results of communication benchmark for the mesh Poisson solver application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

XPROCS × YPROCS grid processes) — as follows First, a high-level decomposition gives us values for $\mathcal{T}_{\text{elapsed}}$ (total estimated elapsed time) and $\mathcal{T}_{\text{process}}$ (total estimated process time, excluding process-setup overhead):

$$\begin{aligned} \mathcal{T}_{\text{elapsed}} &= \mathcal{T}_{\text{process}} + \mathcal{T}_{\text{overhead}} \\ \mathcal{T}_{\text{process}} &= \mathcal{T}_{\text{startup}} + \mathcal{T}_{\text{computation}} + \mathcal{T}_{\text{check}} + \mathcal{T}_{\text{copy}} + \mathcal{T}_{\text{finish}} \end{aligned}$$

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are as follows:

$$\begin{aligned} \mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{init}} \\ \mathcal{T}_{\text{computation}} &= \text{NSTEPS} \times \mathcal{T}_{\text{comp}} \\ \mathcal{T}_{\text{check}} &= \frac{\text{NSTEPS}}{\text{NCHECK}} \times \mathcal{T}_{\text{check\_converge}} \\ \mathcal{T}_{\text{copy}} &= \text{NSTEPS} \times \mathcal{T}_{\text{copy\_values}} \\ \mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{output}} \end{aligned}$$

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines,

as follows:

$$
\begin{aligned}
\mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{init}} + \mathcal{T}_{\text{HtoG}} + \mathcal{T}_{\text{xintersect}} + \mathcal{T}_{\text{yintersect}} + \mathcal{T}_{\text{iglobal}} + \mathcal{T}_{\text{jglobal}} \\
\mathcal{T}_{\text{computation}} &= \text{NSTEPS} \times \left(\frac{\mathcal{T}_{\text{comp}}}{\text{XPROCS} \times \text{YPROCS}} + \mathcal{T}_{\text{update\_bdry}} \right) \\
\mathcal{T}_{\text{check}} &= \frac{\text{NSTEPS}}{\text{NCHECK}} \times \left(\frac{\mathcal{T}_{\text{check\_converge}}}{\text{XPROCS} \times \text{YPROCS}} + \mathcal{T}_{\text{merge\_real\_maxabs}} \right) \\
\mathcal{T}_{\text{copy}} &= \text{NSTEPS} \times \left(\frac{\mathcal{T}_{\text{copy\_values}}}{\text{XPROCS} \times \text{YPROCS}} \right) \\
\mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{GtoH}} + \mathcal{T}_{\text{output}}
\end{aligned}
$$

Experimental Results. For this experiment we used the following values of the program parameters:

$$
\begin{aligned}
\text{NSTEPS} &= 1000 \\
\text{NCHECK} &= 10 \\
\text{XPROCS} &= \text{YPROCS} = \sqrt{n}
\end{aligned}
$$

where n is the number of grid (non-"host") processors (1, 4, 9, 16, 25, or 36). Table 3.6 and Figures 3.4 and 3.5 compare predicted with observed times. For this experiment, predicted times generally agreed fairly well with observed times, with predicted times being, as we expected, somewhat conservative. Our model did less well for this application than for the heat equation application in predicting scalability, but it still came fairly close.

	Time (secs) on n nodes, not including host node					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Expected Elapsed Time	496	149	94	78	91	91
Actual Elapsed Time	522	148	87	69	58	62
Expected Process Time	490	141	80	59	59	54
Actual Process Time	517	140	73	47	34	32

Table 3.6: Execution times for the mesh Poisson solver application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figures 3.4 and 3.5 for corresponding graphs.

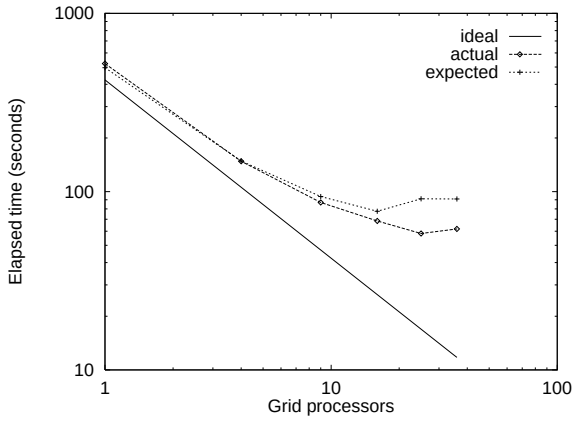


Figure 3.4: Elapsed times for the mesh Poisson solver application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 3.6 for corresponding table.

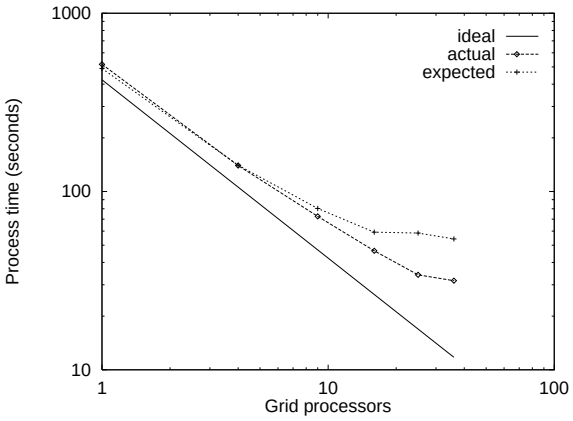


Figure 3.5: Process times for the mesh Poisson solver application, running on the specified number of nodes (plus a "host" node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 3.6 for corresponding table.

Chapter 4

Mesh-Spectral Applications

In an application written using the mesh archetype [DM96]: Data is based on three-dimensional grids (arrays), with one- and two-dimensional grids considered as special cases of three-dimensional grids; a computation may contain multiple grids of different dimensions. Computation consists of some sequence of the following operations:

- Neighbor operations in which new values are computed for each point in a grid based on values at that point and nearby points.
- Row operations, in which new values are computed for each point in a grid based on values in the same row.
- Column operations, defined analogously.
- Reduction operations over a grid (for example., global maximum).

Figures 4.1 and 4.2 illustrate two of these operations (a neighbor operation and a row operation respectively) in a two-dimensional grid.

Mesh-spectral computations are readily parallelized using the following approach: The overall structure of the computation is based on the SPMD (Single Program, Multiple Data) model; that is, it consists of some number N of processes all executing the same program, each on its own data. Each 3-dimensional data grid is distributed over the processes based on a 3-dimensional *process grid* of some or all of the N processes; computation of new values for grid variables is similarly distributed. In the course of a computation, a data grid can be redistributed (that is, the process grid over which it is distributed can be changed); this is usually done when one distribution is convenient for part of the computation and a different distribution is convenient for another part of the computation. Non-grid variables (for example, global constants and reduction variables) are duplicated in each process; their values are kept consistent.

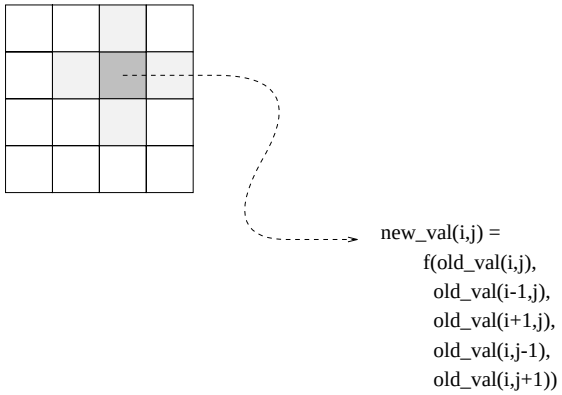


Figure 4.1: Neighbor operation.

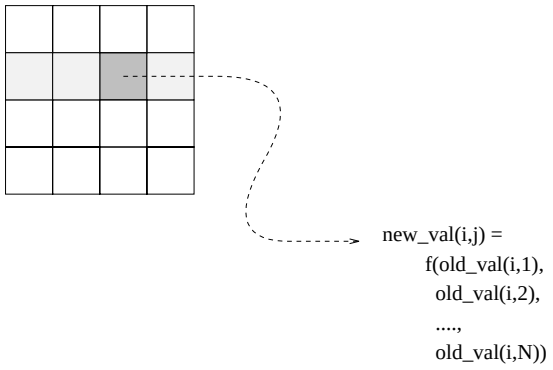


Figure 4.2: Row operation.

This approach is discussed by Massingill and Davis [DM96], including details about how to parallelize sequential code and about how to build an application program from the archetype-provided code template and library. The library includes routines that encapsulate the necessary communication operations (redistribution, boundary exchange, reductions, and broadcast) and a number of utility routines for index manipulation and other housekeeping. This archetype code (template and library) has been implemented in Fortran M and Fortran with MPI. All implementations have essentially the same application program-

mer interface, so applications developed using one implementation are trivially ported to another.

4.1 Two-dimensional Fast Fourier Transform

This application [DM96] performs a two-dimensional FFT in place. As described in *Numerical Recipes* [PFTV86], computing a two-dimensional FFT in place is accomplished by performing a one-dimensional FFT on each row of a two-dimensional array, and then performing a one-dimensional FFT on each column of the resulting two-dimensional array. Optimality of the 2D FFT depends largely on the choice of algorithm (and implementation) for the 1D FFT.

In our application, the actual FFTs are performed using a sequential subroutine library that allows the computation of FFTs on a set of vectors with a single subroutine call. The sequential version of the application is straightforward (first use the library subroutine to perform 1D FFTs on each row, and then use it to perform 1D FFTs on each column. A parallel version based on the mesh-spectral archetype is not much more complicated: In order to perform the FFT computations using the same sequential subroutine library, the program employs two distributions for the two-dimensional array on which the FFT is to be performed: a distribution by rows (which, for each row, puts all data in a single process, allowing it to be processed with a sequential FFT subroutine), and a distribution by columns (which has the same effect on columns). Data is initially distributed by rows; after the FFT-by-rows is performed, it is redistributed by columns and the FFT-by-columns is performed. It is then redistributed by rows before being written out. The FFT computation (calculation by rows, redistribution, calculation by columns, and then redistribution again) is repeated several times to reduce the proportion of total application time spent on I/O.

Our 1D FFT code was written by Clive Temperton for the Meteorological Office in England. The code was originally designed for vector machines like the Cray; today, the library is widely available at all supercomputer centers. Note that the details of the one-dimensional FFT implementation are not relevant to the parallelization, and while they affect performance, we use the same one-dimensional FFT in both the sequential and parallel programs. Thus, overall performance could be improved by choosing a faster 1D FFT, without changing the parallel aspects of the code.

We note that [PFTV86] is not the most efficient implementation of an FFT, but it is well-understood and easily implemented using the mesh-spectral archetype; in this paper, we use this algorithm primarily to illustrate the performance model, rather than attempt to achieve the fastest possible FFT (as is the focus of other work, such as [Win78]). Duhamel and Vetterli provide an excellent survey of FFTs [DV90]. A good comparison of the FFT algorithm we use with more efficient ones (such as a split-radix algorithm) on a vanilla workstation is

given in [Arn96] (although, with a multicomputer, a simpler butterfly structure might be better for more actual computation [Har96]).

Benchmarking. The computational benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate process), $\mathcal{T}_{\text{read\_fft}}$ (set up grid and read input data), $\mathcal{T}_{\text{init}}$ (initialize for FFT), $\mathcal{T}_{\text{row\_fft}}$ (perform FFTs on rows), $\mathcal{T}_{\text{col\_fft}}$ (perform FFTs on columns), and $\mathcal{T}_{\text{write\_fft}}$ (write output data). Results are given in Table 4.1. Observe that results for this benchmark are independent of the choice of archetype implementation.

Measurement	Time (msecs)
$\mathcal{T}_{\text{overhead}}$	2800
$\mathcal{T}_{\text{read\_fft}}$	16161
$\mathcal{T}_{\text{init}}$	3
$\mathcal{T}_{\text{row\_fft}}$	2854
$\mathcal{T}_{\text{col\_fft}}$	3335
$\mathcal{T}_{\text{write\_fft}}$	12934

Table 4.1: Results of computational benchmark for the mesh-spectral 2D FFT application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

The communication benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate processes), $\mathcal{T}_{\text{col\_to\_row}}$ (redistribute data, columns to rows), $\mathcal{T}_{\text{row\_to\_col}}$ (redistribute data, rows to columns), $\mathcal{T}_{\text{data\_widths}}$ (housekeeping), $\mathcal{T}_{\text{local\_pos}}$ (housekeeping), $\mathcal{T}_{\text{set\_mesh}}$ (set up data grid), $\mathcal{T}_{\text{read\_mesh}}$ (perform communication associated with reading data), and $\mathcal{T}_{\text{write\_mesh}}$ (perform communication associated with writing data). We ran this benchmark on 1, 2, 4, 8, 16, and 32 processors. Results are given in Table 4.2. The computational and communication benchmark programs appear in the appendix of [RM96].

Performance Model. We use our performance model and the program, given in the appendix of [RM96], to compute estimated running time in terms of the preceding list of benchmark values and two additional program parameters — NREPEATS (number of times to repeat the FFT) and NPROCS (number of processes) — as follows. First, a high-level decomposition gives us values for $\mathcal{T}_{\text{elapsed}}$ (total estimated elapsed time) and $\mathcal{T}_{\text{process}}$ (total estimated process time, excluding process-setup overhead):

$$\mathcal{T}_{\text{elapsed}} \quad = \quad \mathcal{T}_{\text{process}} + \mathcal{T}_{\text{overhead}}$$

Measurement	Time (msecs) on n nodes					
	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
$\mathcal{T}_{\text{overhead}}$	3930	4430	6030	9400	17130	32180
$\mathcal{T}_{\text{col\_to\_row}}$	3197	1669	660	303	145	61
$\mathcal{T}_{\text{row\_to\_col}}$	3195	1856	1103	1924	1606	1325
$\mathcal{T}_{\text{data\_widths}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_pos}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{set\_mesh}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{read\_mesh}}$	619	435	344	304	287	289
$\mathcal{T}_{\text{write\_mesh}}$	634	423	377	352	350	371

Table 4.2: Results of communication benchmark for the mesh-spectral 2D FFT application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

$$\mathcal{T}_{\text{process}} = \text{NREPEATS} \times (\mathcal{T}_{\text{startup}} + \mathcal{T}_{\text{computation}} + \mathcal{T}_{\text{finish}})$$

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are as follows:

$$\begin{aligned} \mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{read\_fft}} + \mathcal{T}_{\text{init}} \\ \mathcal{T}_{\text{computation}} &= \mathcal{T}_{\text{row\_fft}} + \mathcal{T}_{\text{col\_fft}} \\ \mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{write\_fft}} \end{aligned}$$

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines, as follows:

$$\mathcal{T}_{\text{startup}} = \mathcal{T}_{\text{set\_mesh}} + \mathcal{T}_{\text{read\_fft}} + \mathcal{T}_{\text{read\_mesh}} + \mathcal{T}_{\text{init}}$$

$$\begin{aligned} \mathcal{T}_{\text{computation}} = {} & \mathcal{T}_{\text{local\_pos}} + \mathcal{T}_{\text{data\_widths}} + \left(\frac{\mathcal{T}_{\text{row\_fft}}}{\text{NPROCS}} \right) + \mathcal{T}_{\text{row\_to\_col}} + \\ & \mathcal{T}_{\text{local\_pos}} + \mathcal{T}_{\text{data\_widths}} + \left(\frac{\mathcal{T}_{\text{col\_fft}}}{\text{NPROCS}} \right) + \mathcal{T}_{\text{col\_to\_row}} \end{aligned}$$

$$\mathcal{T}_{\text{finish}} = \mathcal{T}_{\text{write\_fft}} + \mathcal{T}_{\text{write\_mesh}}$$

The computational and communication benchmark programs appear in the appendix of [RM96].

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NREPEATS} = 10$$
$$\text{NPROCS} = n$$

where n is the number of processors (1, 2, 4, 8, 16, or 32). Table 4.3 and Figures 4.3 and 4.4 compare predicted with observed times. For this experiment, predicted times generally agreed well with observed times, with predicted times mostly being, as we expected, somewhat pessimistic. (The exception is the predictions for 4 nodes, which in both cases were optimistic.)

The overall performance of this application is admittedly disappointing. In part this is because the application reads and writes the whole array; including this substantial I/O degrades performance but makes the program slightly more realistic and demonstrates that the performance model is compatible with the archetype's I/O handling. However, it appears that this application simply does not scale very well; for more than a few processors, performance gains obtained by distributing the computation are largely negated by the additional time required for interprocess communication. Possibly this could be overcome by optimizing the archetype library (the current implementation is an unoptimized proof-of-concept version, which could be replaced by an optimized version, thus improving performance of all mesh-spectral applications).

Despite the application's unimpressive performance, however, this experiment validates our model, since predicted execution times were close to actual execution times. A situation such as this one suggests an additional use for a good performance model — deciding on the basis of the model that a particular parallelization scheme is not likely to be effective without actually coding it up and trying it.

4.2 Poisson Solver

This application is another implementation, using the mesh-spectral archetype this time, of the Poisson solver described in Section 3.2, with one additional feature: Values for stepsize (`H`) and convergence tolerance (`TOL`) are to be read at runtime from standard input or an input file. Both sequential and parallel versions are very similar to those described in Section 3.2, except that the parallel version uses the mesh-spectral archetype library rather than the mesh archetype library, and both versions read in stepsize and tolerance. (The parallel version performs this read in the archetype's designated I/O process and then uses `broadcast` to copy their values to the other processes.) We com-

	Time (secs) on n nodes					
	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
Expected Elapsed Time	160	101	69	69	68	78
Actual Elapsed Time	157	100	80	51	49	60
Expected Process Time	156	96	63	60	51	45
Actual Process Time	153	95	73	41	32	29

Table 4.3: Execution times for the mesh-spectral 2D FFT application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Problem size is 800 by 800 points and 10 repetitions. Times are in seconds. See Figures 4.3 and 4.4 for corresponding graphs.

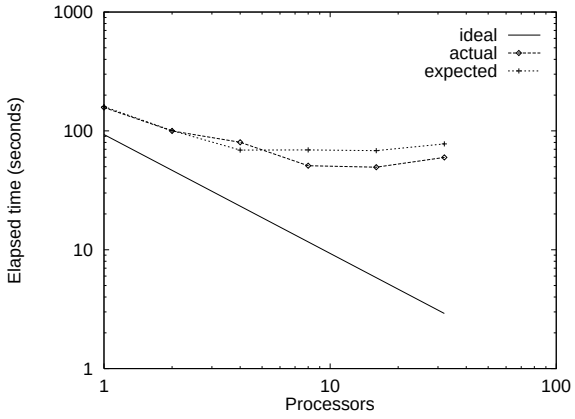


Figure 4.3: Elapsed times for the mesh-spectral 2D FFT application. running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Problem size is 800 by 800 points and 10 repetitions. See Table 4.3 for corresponding table.

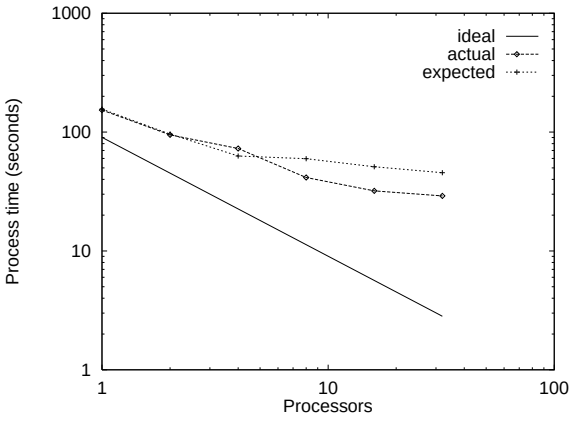


Figure 4.4: Process times for the mesh-spectral 2D FFT application. running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Problem size is 800 by 800 points and 10 repetitions. See Table 4.3 for corresponding table.

pare the performance of the mesh-spectral implementation to that of the mesh implementation in Section 5.1.

Benchmarking. The computational benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate process), $\mathcal{T}_{\text{read\_const}}$ (read constants), $\mathcal{T}_{\text{init}}$ (initialize grid values), $\mathcal{T}_{\text{comp}}$ (calculate new values for all grid points), $\mathcal{T}_{\text{check\_converge}}$ (check for convergence), $\mathcal{T}_{\text{copy\_values}}$ (copy new values back to `uk`), and $\mathcal{T}_{\text{output}}$ (output results). Results are given in Table 4.4. Observe that results for this benchmark are independent of the choice of archetype implementation.

The communication benchmark measures values for the following times: $\mathcal{T}_{\text{overhead}}$ (start and terminate processes), $\mathcal{T}_{\text{set\_mesh}}$ (set up grid), $\mathcal{T}_{\text{blk\_to\_one}}$ (redistribute data, block distribution to all-in-one), $\mathcal{T}_{\text{one\_to\_blk}}$ (redistribute data, all-in-one to block distribution), $\mathcal{T}_{\text{bdry\_exchg}}$ (update ghost boundaries by exchanging data with neighboring processes), $\mathcal{T}_{\text{bcast}}$ (broadcast constants), $\mathcal{T}_{\text{global\_max\_dp}}$ (compute global maximum from local maxima), and housekeeping routines (mostly for manipulating global and local indices) $\mathcal{T}_{\text{data\_bounds}}$, $\mathcal{T}_{\text{intersect}}$, $\mathcal{T}_{\text{local\_pos}}$, $\mathcal{T}_{\text{local\_to\_global}}$, $\mathcal{T}_{\text{pack}}$, and $\mathcal{T}_{\text{unpack}}$. We ran this benchmark on 1, 4, 9, 16, 25, and 36 processors. Results are given in Table 4.5.

The computational and communication benchmark programs appear in the appendix of [RM96].

Measurement	Time (msecs)
$\mathcal{T}_{\text{overhead}}$	100
$\mathcal{T}_{\text{read\_const}}$	15
$\mathcal{T}_{\text{init}}$	222
$\mathcal{T}_{\text{comp}}$	427
$\mathcal{T}_{\text{check\_converge}}$	134
$\mathcal{T}_{\text{copy\_values}}$	62
$\mathcal{T}_{\text{output}}$	15

Table 4.4: Results of computational benchmark for the mesh-spectral Poisson solver application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

	Time (msecs) on n nodes					
Measurement	$n=1$	$n=4$	$n=9$	$n=16$	$n=25$	$n=36$
$\mathcal{T}_{\text{overhead}}$	3850	9630	15950	27380	48330	86150
$\mathcal{T}_{\text{set\_mesh}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{blk\_to\_one}}$	2928	1957	1365	1219	954	1241
$\mathcal{T}_{\text{one\_to\_blk}}$	2951	2474	2051	2026	2222	1792
$\mathcal{T}_{\text{bdry\_exchg}}$	0	11	13	14	16	20
$\mathcal{T}_{\text{bcast}}$	0	4	14	35	75	145
$\mathcal{T}_{\text{data\_bounds}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{global\_max\_dp}}$	0	12	25	53	96	171
$\mathcal{T}_{\text{intersect}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_pos}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_to\_global}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{pack}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{unpack}}$	0	0	0	0	0	0

Table 4.5: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

Performance Model. We use our performance model and the program, given in the appendix of [RM96], to compute estimated running time in terms of the preceding list of benchmark values and a few additional program parameters — NSTEPS (number of loop iterations), NCHECK (frequency of convergence checking), and NXPROCS and NYPROCS (dimensions of process grid, implying a total of NXPROCS × NYPROCS processes) — as follows. First, a high-level decomposition gives us values for $\mathcal{T}_{\text{elapsed}}$ (total estimated elapsed time) and $\mathcal{T}_{\text{process}}$ (total estimated process time, excluding process-setup overhead):

$$\begin{aligned} \mathcal{T}_{\text{elapsed}} &= \mathcal{T}_{\text{process}} + \mathcal{T}_{\text{overhead}} \\ \mathcal{T}_{\text{process}} &= \mathcal{T}_{\text{startup}} + \mathcal{T}_{\text{computation}} + \mathcal{T}_{\text{check}} + \mathcal{T}_{\text{copy}} + \mathcal{T}_{\text{finish}} \end{aligned}$$

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are as follows:

$$\begin{aligned} \mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{read\_const}} + \mathcal{T}_{\text{init}} \\ \mathcal{T}_{\text{computation}} &= \text{NSTEPS} \times \mathcal{T}_{\text{comp}} \\ \mathcal{T}_{\text{check}} &= \frac{\text{NSTEPS}}{\text{NCHECK}} \times \mathcal{T}_{\text{check\_converge}} \\ \mathcal{T}_{\text{copy}} &= \text{NSTEPS} \times \mathcal{T}_{\text{copy\_values}} \\ \mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{output}} \end{aligned}$$

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines, as follows:

$$\begin{aligned} \mathcal{T}_{\text{startup}} &= \mathcal{T}_{\text{local\_pos}} + \mathcal{T}_{\text{read\_const}} + \mathcal{T}_{\text{bcast}} + \mathcal{T}_{\text{set\_mesh}} + \mathcal{T}_{\text{local\_pos}} + \\ &\quad \mathcal{T}_{\text{init}} + \mathcal{T}_{\text{one\_to\_blk}} + \mathcal{T}_{\text{local\_pos}} \\ \mathcal{T}_{\text{computation}} &= (\mathcal{T}_{\text{pack}} \times 3) + (\mathcal{T}_{\text{unpack}} \times 3) + \\ &\quad \mathcal{T}_{\text{data\_bounds}} + \mathcal{T}_{\text{intersect}} + \mathcal{T}_{\text{local\_to\_global}} + \\ &\quad \text{NSTEPS} \times \left(\frac{\mathcal{T}_{\text{comp}}}{\text{XPROCS} \times \text{YPROCS}} + \mathcal{T}_{\text{bdry\_exchg}} \right) \\ \mathcal{T}_{\text{check}} &= \frac{\text{NSTEPS}}{\text{NCHECK}} \times \left(\frac{\mathcal{T}_{\text{check\_converge}}}{\text{XPROCS} \times \text{YPROCS}} + \mathcal{T}_{\text{global\_max\_dp}} \right) \\ \mathcal{T}_{\text{copy}} &= \text{NSTEPS} \times \left(\frac{\mathcal{T}_{\text{copy\_values}}}{\text{NXPROCS} \times \text{NYPROCS}} \right) \\ \mathcal{T}_{\text{finish}} &= \mathcal{T}_{\text{output}} \end{aligned}$$

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NSTEPS} = 1000$$
$$\text{NCHECK} = 10$$
$$\text{NXPROCS} = \text{NYPROCS} = \sqrt{n}$$

where n is the number of processors (1, 4, 9, 16, 25, or 36). Table 4.6 and Figures 4.5 and 4.6 compare predicted with observed times. For this experiment, predicted times agreed well with observed times. Surprisingly, several predictions were slightly optimistic, but overall the agreement was quite good for this application. Our model also correctly predicts the scalability of the application, validating its utility in helping programmers choose granularity.

	Time (secs) on n nodes					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Expected Elapsed Time	512	152	91	81	98	142
Actual Elapsed Time	521	151	91	78	95	130
Expected Process Time	508	142	75	54	49	55
Actual Process Time	516	143	75	54	55	63

Table 4.6: Execution times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds. See Figures 4.5 and 4.6 for corresponding graphs.

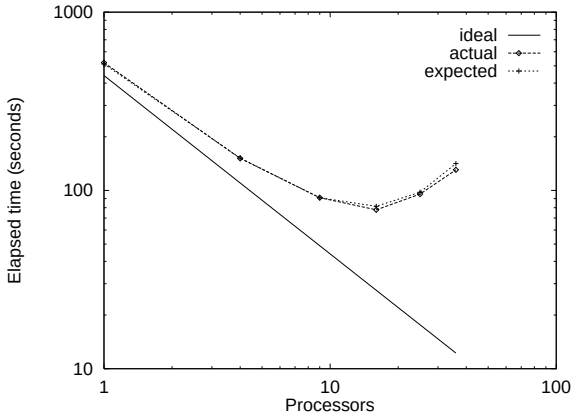


Figure 4.5: Elapsed times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds. See Table 4.6 for corresponding table.

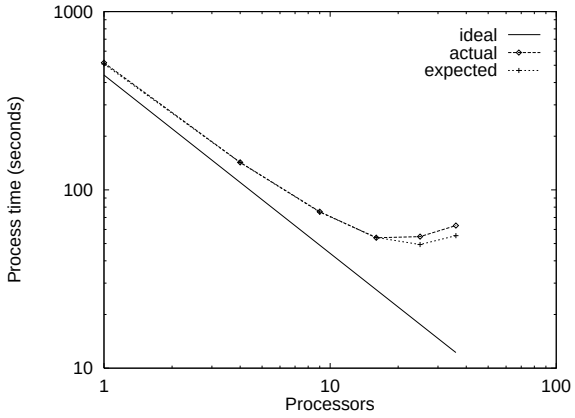


Figure 4.6: Process times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds. See Table 4.6 for corresponding table.

Chapter 5

Evaluating the Performance Model

In this section, we evaluate the utility of our performance model. Through selected experiments, we show that our performance analysis works well when applied to different archetypes (Section 5.1), to different architectures (Section 5.2), and to different communication libraries (Section 5.3). In addition, we show that our performance model can be used to choose between different data distributions (Section 5.4), and that our performance model can be used to simulate actual program executions to predict expected running times (Section 5.5).

5.1 Performance Analysis Across Archetypes

In this section, we demonstrate that our performance analysis works well when used to compare running times for the same application developed using different archetypes. For this experiment, we employ two versions of the Poisson solver application (described in Sections 3.2 and 4.2), both implemented in Fortran M and running on an IBM SP2, but one using the mesh archetype and one the mesh-spectral archetype.

Benchmarking. The computational and communication benchmark programs for this application appear in the appendix of [RM96]. Results of the computational and communication benchmarks for the mesh version of the application, executed using the target archetype implementation and architecture, appear in Tables 3.4 and 3.5, respectively. Results of the computational and communication benchmarks for the mesh-spectral version of the application, executed using the target archetype implementation and architecture, appear in Tables 4.4 and 4.5, respectively.

Performance Models. Our performance model for the mesh Poisson solver application is given in Section 3.2, based on the program in the appendix of [RM96]. Our performance model for the mesh-spectral Poisson solver application is given in Section 4.2, based on the program in the appendix of [RM96].

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NSTEPS} = 1000$$
$$\text{NCHECK} = 10$$

For the mesh version, we used

$$\text{XPROCS} = \text{YPROCS} = \sqrt{n}$$

where n is the number of grid (non-"host") processors (1, 4, 9, 16, 25, or 36). For the mesh-spectral version, we used

$$\text{NXPROCS} = \text{NYPROCS} = \sqrt{n}$$

where n is the number of processors (1, 4, 9, 16, 25, or 36). Table 5.1 and Figure 5.1 compare elapsed times (predicted and observed) for the two programs. Table 5.2 and Figure 5.2 compare process times (predicted and observed) for the two programs. These results have been discussed previously (in Sections 3.2 and 4.2); note again that for both versions of the application the model's predictions about execution times and scaling are generally good. The model also correctly predicts that overall the mesh-spectral version of the application performs better.

5.2 Performance Analysis Across Architectures

In this section, we demonstrate that our performance analysis works well when used to compare running times for the same application developed using the same archetype, executing on different target machines (with portability as an automatic consequence of using a portable archetype implementation). For this experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), implemented in Fortran with MPI, running on an IBM SP2 and on a network of 166 MHz Pentium personal computers connected by 100 Mbps Ethernet.

Benchmarking. The computational and communication benchmark programs for this application appear in the appendix of [RM96]. Applying our performance analysis for a particular implementation and architecture requires results from executing both benchmarks on an appropriate system. For the

	Time (secs) on n nodes, not including host node					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Mesh-Spectral Expected Elapsed Time	512	152	91	81	98	142
Mesh-Spectral Actual Elapsed Time	521	151	91	78	95	130
Mesh Expected Elapsed Time	496	149	94	78	91	91
Mesh Actual Elapsed Time	522	148	87	69	58	62

Table 5.1: Elapsed times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a "host" node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 5.1 for corresponding graph.

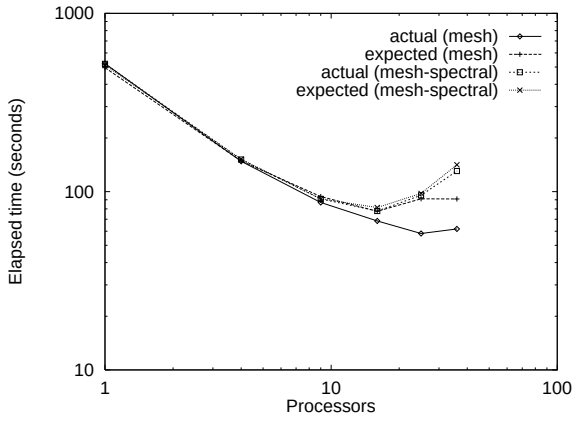


Figure 5.1: Elapsed times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a "host" node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.1 for corresponding table.

	Time (secs) on n nodes, not including host node					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Mesh-Spectral Expected Process Time	508	142	75	54	49	55
Mesh-Spectral Actual Process Time	516	143	75	54	55	63
Mesh Expected Process Time	490	141	80	59	59	54
Mesh Actual Process Time	517	140	73	47	34	32

Table 5.2: Process times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a "host" node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 5.2 for corresponding graph.

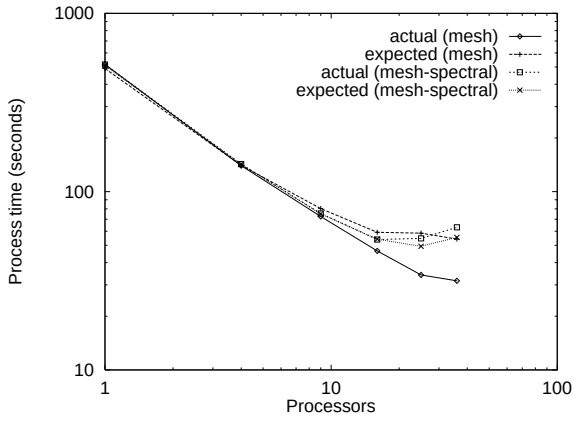


Figure 5.2: Process times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a "host" node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.2 for corresponding table.

SP2, we can reuse the computational benchmark results shown in Table 4.4, since the target architecture is the same. We must, however, rerun the communication benchmark using the MPI-based archetype implementation. As before, we ran this benchmark on 1, 4, 9, 16, 25, and 36 processors; results are given in Table 5.3. For the network of Pentiums, we must rerun both computational and communication benchmarks. Results of running the computational benchmark on one Pentium processor appear in Table 5.4. Due to a bug in the MPI implementation installed on our target network, we were unable to make use of more than 9 processors, so we ran the communication benchmark for 1, 2, 4, 6, 8, and 9 processors. Results are given in Table 5.5.

	Time (msecs) on n nodes					
Measurement	$n=1$	$n=4$	$n=9$	$n=16$	$n=25$	$n=36$
$\mathcal{T}_{\text{overhead}}$	3930	6030	10280	17130	26400	39130
$\mathcal{T}_{\text{set\_mesh}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{blk\_to\_one}}$	3218	2235	1374	1276	1172	1026
$\mathcal{T}_{\text{one\_to\_blk}}$	3243	1977	1829	1666	1750	1698
$\mathcal{T}_{\text{bdry\_exchg}}$	0	4	4	4	4	5
$\mathcal{T}_{\text{bcast}}$	0	0	1	1	2	3
$\mathcal{T}_{\text{data\_bounds}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{global\_max\_dp}}$	0	1	1	2	3	4
$\mathcal{T}_{\text{intersect}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_pos}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_to\_global}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{pack}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{unpack}}$	0	0	0	0	0	0

Table 5.3: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

Performance Model. Our performance model for this application is given in Section 4.2, based on the program in the appendix of [RM96], and is applicable to both architectures.

Measurement	Time (msecs)
$\mathcal{T}_{\text{overhead}}$	150
$\mathcal{T}_{\text{read\_const}}$	5
$\mathcal{T}_{\text{init}}$	271
$\mathcal{T}_{\text{comp}}$	941
$\mathcal{T}_{\text{check\_converge}}$	265
$\mathcal{T}_{\text{copy\_values}}$	244
$\mathcal{T}_{\text{output}}$	40

Table 5.4: Results of computational benchmark for the mesh-spectral Poisson solver application, running on a single 166 MHz Pentium using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

	Time (msecs) on n nodes					
Measurement	$n=1$	$n=2$	$n=4$	$n=6$	$n=8$	$n=9$
$\mathcal{T}_{\text{overhead}}$	3000	7530	9930	14830	18000	18150
$\mathcal{T}_{\text{set\_mesh}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{blk\_to\_one}}$	3957	3805	3307	2310	2243	2133
$\mathcal{T}_{\text{one\_to\_blk}}$	4115	3738	4563	4961	5230	5134
$\mathcal{T}_{\text{bdry\_exchg}}$	0	11	15	19	209	296
$\mathcal{T}_{\text{bcast}}$	0	0	2	3	5	5
$\mathcal{T}_{\text{data\_bounds}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{global\_max\_dp}}$	0	2	4	6	9	12
$\mathcal{T}_{\text{intersect}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_pos}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{local\_to\_global}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{pack}}$	0	0	0	0	0	0
$\mathcal{T}_{\text{unpack}}$	0	0	0	0	0	0

Table 5.5: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on a network of 166 MHz Pentiums using Fortran with MPI, communicating over 100 Mbps Ethernet. Grid size is 800 by 800 points. Times are in milliseconds.

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NSTEPS} = 1000$$
$$\text{NCHECK} = 10$$

On the SP2, we used

$$\text{NXPROCS} = \text{NYPROCS} = \sqrt{n}$$

where n is the number of processors (1, 4, 9, 16, 25, or 36). On the network of Pentiums, we used the following values of (NXPROCS, NYPROCS): (1,1), (1,2), (2,2), (2,3), (2,4), and (3,3), corresponding to 1, 2, 4, 8, and 9 processors. Table 5.6 and Figure 5.3 compare elapsed times (predicted and observed) for the two architectures (network of Pentiums and IBM SP2). Table 5.7 and Figure 5.4 compare process times (predicted and observed) for the two architectures. For this experiment, predicted times agreed well with observed times for both architectures. Surprisingly, many predicted times for the network of Pentiums were optimistic, though not extremely so. For both architectures, our model predicts the scalability of the application pretty well, and it correctly predicts the expected performance difference between the two architectures.

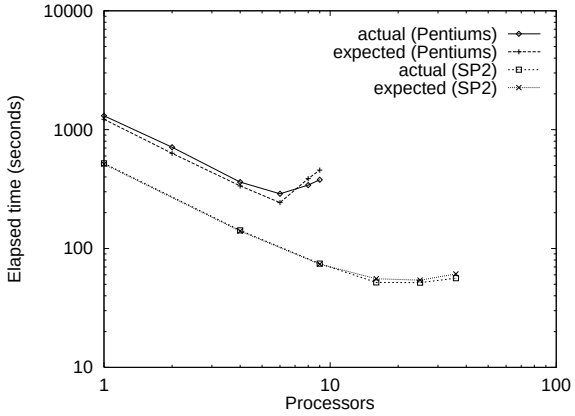


Figure 5.3: Elapsed times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.6 for corresponding table.

	Time (secs) on n nodes				
	$n = 1$	$n = 2$	$n = 4$	$n = 6$	$n = 8$
SP2 Expected Elapsed Time	513	–	140	–	–
SP2 Actual Elapsed Time	520	–	142	–	–
Pentium Expected Elapsed Time	1222	632	337	243	387
Pentium Actual Elapsed Time	1308	712	362	288	342

	Time (secs) on n nodes			
	$n = 9$	$n = 16$	$n = 25$	$n = 36$
SP2 Expected Elapsed Time	74	56	54	61
SP2 Actual Elapsed Time	75	52	52	56
Pentium Expected Elapsed Time	457	–	–	–
Pentium Actual Elapsed Time	379	–	–	–

Table 5.6: Elapsed times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 5.3 for corresponding graph.

5.3 Performance Analysis Across Libraries

In this section, we demonstrate that our performance analysis works well when used to compare running times for the same application developed using the same archetype for the same target machine, using different communication libraries (i.e., different archetype implementations). For this experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), comparing a Fortran M implementation running on an IBM SP2 with an MPI implementation also running on an IBM SP2.

Benchmarking. The computational and communication benchmark programs appear in the appendix of [RM96]. As in Section 5.2, applying our performance analysis for a particular implementation requires results from executing both benchmarks on appropriate system. Since here we are comparing different archetype implementations on the same target architecture, we can use the same computational benchmark results for both, namely the ones presented in Table 4.4. For the communication benchmark, we need results

	Time (secs) on n nodes				
	$n = 1$	$n = 2$	$n = 4$	$n = 6$	$n = 8$
SP2 Expected Process Time	509	–	134	–	–
SP2 Actual Process Time	516	–	135	–	–
Pentium Expected Process Time	1219	625	327	229	370
Pentium Actual Process Time	1305	632	354	276	328

	Time (secs) on n nodes			
	$n = 9$	$n = 16$	$n = 25$	$n = 36$
SP2 Expected Process Time	64	38	28	22
SP2 Actual Process Time	64	39	27	24
Pentium Expected Process Time	439	–	–	–
Pentium Actual Process Time	363	–	–	–

Table 5.7: Process times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 5.4 for corresponding graph.

from executing versions based on both the Fortran M and MPI archetype implementations, presented respectively in Tables 4.5 and 5.3.

Performance Model. Our performance model for this application is given in Section 4.2, based on the program in the appendix of [RM96], and is applicable to both implementations.

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NSTEPS} = 1000$$
$$\text{NCHECK} = 10$$
$$\text{NXPROCS} = \text{NYPROCS} = \sqrt{n}$$

where n is the number of processors (1, 4, 9, 16, 25, or 36). Table 5.8 and Figure 5.5 compare elapsed times (predicted and observed) for the two implementations. Table 5.9 and Figure 5.6 compare process times (predicted and

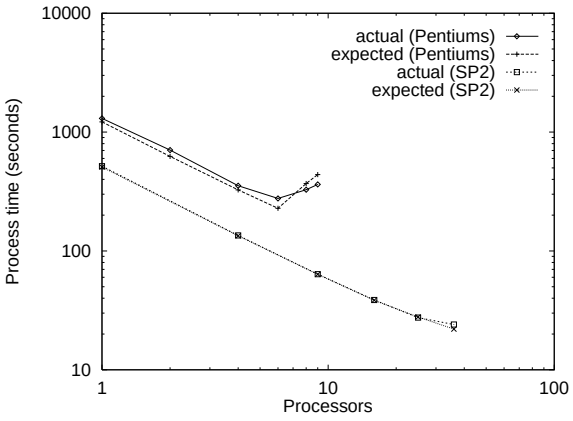


Figure 5.4: Process times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.7 for corresponding table.

observed) for the two implementations. These results have been discussed previously (in Section 4.2 for the Fortran M implementation and Section 5.2 for the MPI implementation); note again that for both implementations the model's predictions about execution times and scaling are generally good. The model also correctly predicts that the MPI implementation performs better than the Fortran M implementation, suggesting that it could help programmers decide between archetype implementations without trying both.

5.4 Performance Analysis and Data Distributions

In this section, we show that our performance model can be used to predict how performance is affected by data distribution and thus help the programmer to choose an efficient data distribution. For our experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), implemented in Fortran with MPI and running on an IBM SP2.

	Time (secs) on n nodes					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Fortran M Expected Elapsed Time	512	152	91	81	98	142
Fortran M Actual Elapsed Time	521	151	91	78	95	130
MPI Expected Elapsed Time	513	140	74	57	54	61
MPI Actual Elapsed Time	520	142	75	52	52	56

Table 5.8: Elapsed times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 5.5 for corresponding graph.

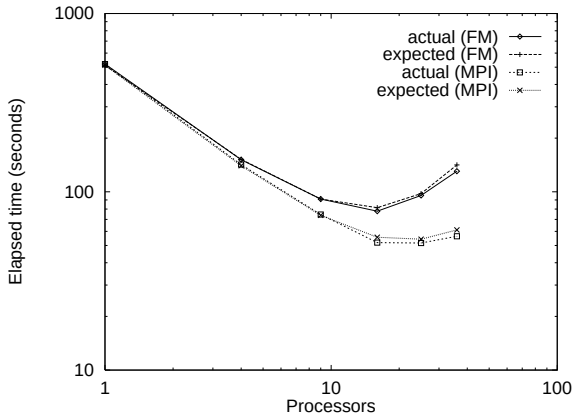


Figure 5.5: Elapsed times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.8 for corresponding table.

	Time (secs) on n nodes					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Fortran M Expected Process Time	508	142	75	54	49	55
Fortran M Actual Process Time	516	143	75	54	55	63
MPI Expected Process Time	509	134	64	39	28	22
MPI Actual Process Time	516	135	64	39	27	24

Table 5.9: Process times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 5.6 for corresponding graph.

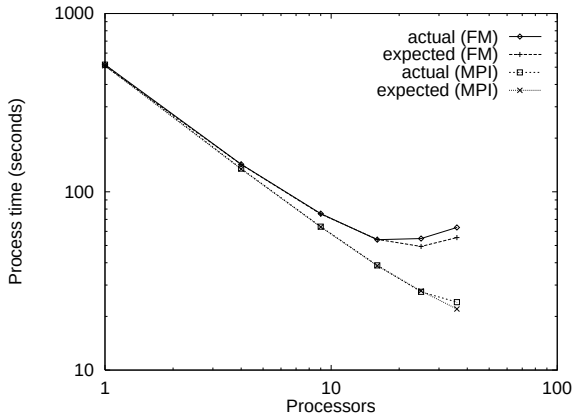


Figure 5.6: Process times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.9 for corresponding table.

Benchmarking. The computational and communication benchmark programs appear in the appendix of [RM96]. Different choices of NXPROCS and NYPROCS (the dimensions of the process grid) imply different data distributions; for example, if NXPROCS = 1, data is distributed by columns. To model the effect of varying the data distribution in this way, we can reuse the computational benchmark results in Table 4.4, but we must rerun the communication benchmark for each choice of (NXPROCS, NYPROCS). We ran the communication benchmark for the following configurations of (NXPROCS, NYPROCS): (1,16), (2,8), (4,4), (8,2), and (16,1). Results are given in Table 5.10.

(NXPROCS, NYPROCS)	(1, 16)	(2, 8)	(4, 4)	(8, 2)	(16, 1)
$\mathcal{T}_{\text{set\_mesh}}$	0	0	0	0	0
$\mathcal{T}_{\text{blk\_to\_one}}$	868	1454	1276	1315	1428
$\mathcal{T}_{\text{one\_to\_blk}}$	1326	1714	1666	1669	1629
$\mathcal{T}_{\text{bdry\_exchg}}$	4	4	4	4	6
$\mathcal{T}_{\text{bcast}}$	1	1	1	1	1
$\mathcal{T}_{\text{data\_bounds}}$	0	0	0	0	0
$\mathcal{T}_{\text{global\_max\_dp}}$	2	2	2	2	2
$\mathcal{T}_{\text{intersect}}$	0	0	0	0	0
$\mathcal{T}_{\text{local\_pos}}$	0	0	0	0	0
$\mathcal{T}_{\text{local\_to\_global}}$	0	0	0	0	0
$\mathcal{T}_{\text{pack}}$	0	0	0	0	0
$\mathcal{T}_{\text{unpack}}$	0	0	0	0	0

Table 5.10: Results of communication benchmark for the mesh-spectral Poisson solver application, running with the specified data distributions on the IBM SP2 using Fortran and MPI, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

Performance Model. Our performance model for this application is given in Section 4.2, based on the program in the appendix of [RM96], and is applicable to all data distributions (parameterized by NXPROCS and NYPROCS).

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NSTEPS} = 1000$$
$$\text{NCHECK} = 10$$

and NXPROCS and NYPROCS as described earlier. Table 5.11 compares predicted with observed times. Again, predicted times agree well overall with actual times. Somewhat surprisingly, the model also predicts that for this application the choice of data distribution has little effect on execution time; nevertheless, this is borne out by observed execution times, suggesting that the model's predictions can indeed help guide the programmer's choice of data distribution.

(NXPROCS, NYPROCS)	(1, 16)	(2, 8)	(4, 4)	(8, 2)	(16, 1)
Expected Process Time	38	39	39	39	41
Actual Process Time	38	38	39	39	41
Expected Elapsed Time	55	56	55	56	58
Actual Elapsed Time	52	52	52	57	57

Table 5.11: Execution times for the mesh-spectral Poisson solver application, running with the specified data distributions on the IBM SP2 using Fortran and MPI, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds.

5.5 Performance Analysis and Simulation

In this section, we show that our performance model can be used to simulate the actual program executions. For our experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), implemented in Fortran M, running on the IBM SP2.

Benchmarking. The computational and communication benchmark programs appear in the appendix of [RM96]. For this experiment, we need not only the average results presented Tables 4.4 and 4.5 but also, for each measurement, the maximum and minimum of the values used to compute the average. These values are shown in Tables 5.12 and 5.13. For each measurement, the simulation generates a uniform distribution using these minimum and maximum values.

Performance Model. Our simulation is based on the performance model given in Section 4.2, and on the program in the appendix of [RM96].

Measurement	Min, max times (msecs)
$\mathcal{T}_{\text{overhead}}$	100, 100
$\mathcal{T}_{\text{read\_const}}$	11, 24
$\mathcal{T}_{\text{init}}$	222, 223
$\mathcal{T}_{\text{comp}}$	426, 431
$\mathcal{T}_{\text{check\_converge}}$	133, 138
$\mathcal{T}_{\text{copy\_values}}$	60, 66
$\mathcal{T}_{\text{output}}$	15, 16

Table 5.12: Results of computational benchmark for the mesh-spectral Poisson solver application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

Experimental Results. For this experiment we used the following values of the program parameters:

$$\begin{aligned} \text{NSTEPS} &= 1000 \\ \text{NCHECK} &= 10 \\ \text{NXPROCS} &= \text{NYPROCS} = \sqrt{n} \end{aligned}$$

where n is the number of processors (1, 4, 9, 16, 25, or 36). Table 5.14 and Figures 5.7 and 5.8 compare predicted, simulated, and actual execution times. These results indicate that our simulation helps us as developers to predict the performance times about as accurately as the performance model. Although before this experiment we surmised that analysis using closed-form equations would be most useful for calculating worst-case performance, and simulation would be most useful for estimating average-case performance, the nature of this application is such that both techniques are useful for estimating actual performance.

Measurement	Min, max times (msecs) on n nodes		
	$n = 1$	$n = 4$	$n = 9$
$\mathcal{T}_{\text{overhead}}$	3300, 5000	7800, 11600	14500, 18300
$\mathcal{T}_{\text{set\_mesh}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{blk\_to\_one}}$	2856, 3006	1916, 2011	1335, 1392
$\mathcal{T}_{\text{one\_to\_blk}}$	2886, 3021	2363, 2603	2016, 2085
$\mathcal{T}_{\text{bdry\_exchg}}$	0, 0	11, 11	13, 13
$\mathcal{T}_{\text{bcast}}$	0, 0	4, 4	14, 14
$\mathcal{T}_{\text{data\_bounds}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{global\_max\_dp}}$	0, 0	12, 12	25, 25
$\mathcal{T}_{\text{intersect}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{local\_pos}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{local\_to\_global}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{pack}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{unpack}}$	0, 0	0, 0	0, 0

Measurement	Min, max times (msecs) on n nodes		
	$n = 16$	$n = 25$	$n = 36$
$\mathcal{T}_{\text{overhead}}$	22000, 40100	41100, 61300	78100, 101900
$\mathcal{T}_{\text{set\_mesh}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{blk\_to\_one}}$	1183, 1243	155, 1238	1082, 1689
$\mathcal{T}_{\text{one\_to\_blk}}$	1951, 2080	2053, 2677	2166, 2212
$\mathcal{T}_{\text{bdry\_exchg}}$	13, 14	16, 17	20, 21
$\mathcal{T}_{\text{bcast}}$	35, 36	74, 76	138, 149
$\mathcal{T}_{\text{data\_bounds}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{global\_max\_dp}}$	53, 55	95, 97	163, 176
$\mathcal{T}_{\text{intersect}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{local\_pos}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{local\_to\_global}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{pack}}$	0, 0	0, 0	0, 0
$\mathcal{T}_{\text{unpack}}$	0, 0	0, 0	0, 0

Table 5.13: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

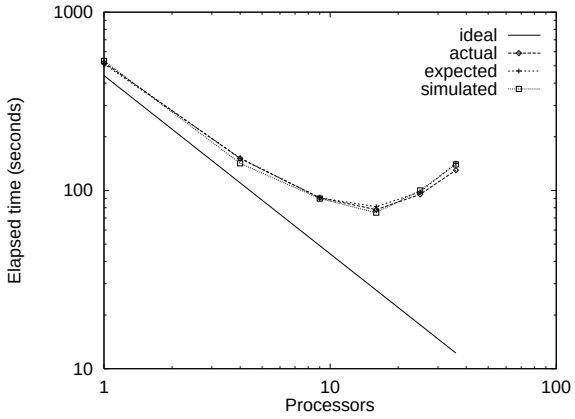


Figure 5.7: Elapsed times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2, without the crossbar switch, using Fortran M. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.14 for corresponding table.

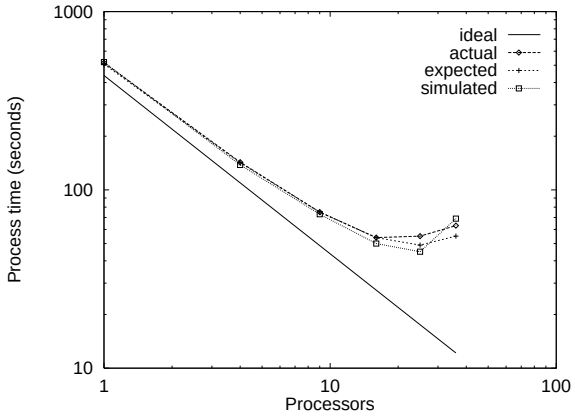


Figure 5.8: Process times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2, without the crossbar switch, using Fortran M. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 5.14 for corresponding table.

	Time (secs) on n nodes					
	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 25$	$n = 36$
Expected Elapsed Time	512	152	91	81	98	142
Simulated Elapsed Time	533	142	90	75	100	140
Actual Elapsed Time	521	151	91	78	95	130
Expected Process Time	508	142	75	54	49	55
Simulated Process Time	521	138	73	50	45	69
Actual Process Time	516	143	75	54	55	63

Table 5.14: Elapsed times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2, without the crossbar switch, using Fortran M. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figures 5.7 and 5.8 for corresponding graphs.

Chapter 6

Conclusions

Our experimental results in Section 5 confirm that execution times as predicted by our model are reasonably close to observed execution times. In addition, experiments designed to test our model's ability to predict how various "tuning choices" affect execution time gave encouraging results: The tuning choices that suggested the best predicted times also gave the best observed times.

The goal of our performance evaluation methodology was to estimate program execution time with sufficient accuracy to guide programmers in making tuning decisions, and to do this in a way that can be incorporated into the application development process relatively easily. Based on the experiments described in Chapter 5, we believe that we have met that goal and conclude that the model, though simple, is practical to the archetype-driven development of applications. Future work could investigate a wider range of archetypes and the applicability of the model to shared-memory architectures.

6.1 Accuracy of Performance Model

Actual and expected running times were reasonably close when computation time was more than half of the overall running time, and when a finer grain of benchmarking was used. In many cases, predicted times were conservative (greater than observed execution times) because our performance model uses implicit barriers, as illustrated by the explanation of Figure 2.2.

Due to time constraints, we did not pursue the question of why in a few cases predicted times were optimistic. One explanation is that our assumptions about repeatability were not valid: For example, we assumed that execution times on the IBM SP2 would not be affected by other users of the machine, since we had exclusive control of the particular nodes we were using; this assumption ignores possible contention for machine-wide communication resources.

6.2 Usefulness of Performance Model

As demonstrated in Chapter 5, the model can help programmers choose between different data partitioning and granularity strategies. We conclude that the model, though simple, can be of use to an application developer.

6.3 Problem Solving Environments

The performance tools we have described in this paper — including the methodology encapsulated by the workflow in Figure 2.1, the analytic models, and the accompanying simulation techniques — can be nicely managed when bundled with a Problem Solving Environment (PSE). According to Gallopoluous et al. [GHR94], PSEs provide "a framework that is all things to all people: they solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science." As demonstrated by Cheng and Fox [CF96], integrating parallel programming paradigms in a software environment (in their example, programming the CM5 using the visualization software AVS) enables application developers to become more productive through useful provided tools.

PSEs represent a way to package a computational solution to a problem with the set of tools and methodologies. Employing these tools and methodologies, a scientist can formulate a problem, solve the problem, optimize the solution through refinement, and analyze the results. The PSE provides a user-friendly environment that is natural to the problem domain; for example, the PSE for Air Quality Models [DC97] provides an integrated user interface for defining, solving, and evaluating smog models. This PSE is integrated with the $3D + T + M^k$ archetype; this suggests that integrating it with the mesh and mesh-spectral archetypes would yield similar benefits. Additionally, the integration of the performance model with the problem solving environment further helps an application developer use an archetype.

This suggests another path for future work as well. The performance model and tools presented in this paper could be applied to other archetypes as well, including the one-deep divide and conquer archetype [MC96], which is a pattern that solves divide and conquer programs while only taking the recursion to a single depth. In addition, the performance analysis work done for the aforementioned air quality models [DM97] could be applied to the framework presented in this paper.

6.4 Shared-Memory Architectures

Most of our experiments with archetype-based application development have targeted distributed-memory architectures. A major advantage of an archetype-based approach to developing applications for such archetypes is that a parallel

programming archetype can specifically address one of the things that makes such applications difficult to develop, namely the distribution of data. This advantage could be equally useful for a shared-memory architecture for which data locality is crucial to performance — i.e., a shared-memory architecture that is most effectively used by treating it as a distributed-memory architecture. It would be relatively easy to port existing archetype implementations that use message-passing to such a machine, after which some of the experiments described in this and other papers on archetypes [CMMM95, MC96] could be repeated.

Whether an archetype-based approach has similar benefits when the target architecture supports a shared-memory model without performance penalties is a more difficult question. Massingill [Mas98] describes an archetype-based approach to application development one of whose stages can be converted in a straightforward way to a program for a shared-memory architecture, but again the experimental work focuses on the later stages of the process, whereby the original algorithm becomes a program for a distributed-memory architecture. Future work could experiment with using this process to develop practical applications for shared-memory architectures.

6.5 Task-Parallel Problems

The performance model in this paper was used to evaluate different data partitions and distributions in SPMD programs written using archetypes. Future work could extend to evaluating the model's utility in analyzing different decompositions and mappings of task-parallel programs as well.

6.6 Event-Oriented Problems

The overall goal of any distributed resource management system is the efficient matching of resource providers and requestors. Using events as our solutions' communication substrate, we can develop distributed control announce-listen algorithms that are both scalable and fault-tolerant [Sch96]. The announce-listen paradigm is used at the messaging layer to assist in resource location, reservation, and scheduling [RRDC97]. We have implemented this messaging facility in Java as *global events*.

Java Beans provide *local events* as a mechanism by which a component informs other components that something interesting has happened. These events can be thought of as active messages; for example, a button is pressed at a source, and channeled through an event listener, to trigger a method in an event observer automatically. An event propagates from an *event source* through an *event notifier* to one or more *event observers* that respond to the events as they arrive. The notifier routes the event to the observers using a

control list, and observers can ask the notifier to be added or removed from this list without notifying the event source.

We have developed a global event structuring mechanism [CRS98] that is identical to the local event model of Java Beans, except that instead of Java Beans' referencing an object within a single Java Virtual Machine, we use a global name for the object, employing the Web's URL convention. Furthermore, because the components of the global event system are distributed, multicast can be used for efficient group communication, instead of Java Beans' local event point-to-point casting.

Using global events, an event is *announced* by a source object in one virtual machine, and notifiers for that event in other virtual machines anywhere on the Internet *listen* for the event and forward it to the appropriate (distributed) observers. Unlike the group communication in *virtual synchrony* [BvR94], it is not necessary for the event sources to know at any point who the event observers will be. Our global event model is useful not only to distribute events and the objects that use them, but also to compose event notifiers, to filter using predicates, and to provide security using access control lists at the event notifier level.

There are several advantages to using global events and soft state. The announce-listen paradigm is fault-resilient [Sch96]; that is, if a resource provider goes away, the system adapts dynamically to continue to meet the requests of the consumers. Furthermore, systems constructed using global events and multicast are compositional and scalable; providers and consumers can add or remove themselves at any point dynamically. Unfortunately, such systems also have the potential for oscillation; that is, if state changes faster than the communication updates, then soft state may give a bad estimate of the current system state.

We are currently investigating the tradeoffs between soft state and hard invariants, between pushing and pulling resource requests and responses, and between a hierarchy of middlemen and a flat requestor-provider structure. These are being explored in the context of the Infospheres Infrastructure [CR97, CKRZ97]. The performance tools we have described in this paper — including the methodology encapsulated by the workflow in Figure 2.1, the analytic models, and the accompanying simulation techniques — can be used in conjunction with distributed programs communicating using events as the messaging facility, and this is an exciting area worthy of future exploration.

Bibliography

[Arn96] J. Arndt. *http://www.spektracom.de/ãrndt/fxt/fftbench.txt*. Available on the Web, 1996.

[BBC+93] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.

[BH93a] P. Brinch Hansen. Model Programs for Computational Science: A Programming Methodology for Multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.

[BH93b] P. Brinch Hansen. Parallel Cellular Automata: A Model Program for Computational Science. *Concurrency: Practice and Experience*, 5(5):425–448, 1993.

[BL94] R. M. Butler and E. L. Lusk. Monitors, Messages, and Clusters—The p4 Parallel Programming System. *Parallel Computing*, 20:547–564, 1994.

[BvR94] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[CF96] G. Cheng and G. C. Fox. Integrating Multiple Parallel Programming Paradigms in a Dataflow-Based Software Environment. *Concurrency: Practice and Experience*, 8(9):667–684, 1996.

[Cha94] K. M. Chandy. Concurrent Program Archetypes. In *Proceedings of the Scalable Parallel Library Conference*, 1994.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[CKRZ97] K. M. Chandy, J. Kiniry, A. Rifkin, and D. Zimmerman. Webs of Archived Distributed Computations for Asynchronous Collaboration. *Journal of Supercomputing*, 11(3):101–118, 1997.

[CMMM95] K. M. Chandy, R. Manohar, B. L. Massingill, and D. I. Meiron. Integrating Task and Data Parallelism with the Collective Communication Archetype. In *Proceedings of the International Parallel Processing Symposium IPPS-9*, April 1995.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[CQ93] M. J. Clement and M. J. Quinn. Analytical Performance Prediction on Multicomputers. In *Proceedings of Supercomputing 93*, November 1993.

[CR97] K. M. Chandy and A. Rifkin. Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions. *Oxford University Press Computer Journal*, 40(8):465–478, October 1997.

[CRS98] K. M. Chandy, A. Rifkin, and E. M. Schooler. Using Announce-Listen with Global Events to Develop Distributed Control Systems. *Concurrency: Practice and Experience*, 10(11–13):1021–1027, 1998.

[DC97] D. Dabdub and K. M. Chandy. A PSE for Air Quality Models Using the 3D + T + M^k Archetype. In *Proceedings of the Air and Waste Management Symposium*, 1997.

[DM96] G. Davis and B. L. Massingill. *The Mesh-Spectral Archetype*. Technical Report CS-TR-96-26, Computer Science Department, California Institute of Technology, 1996.

[DM97] D. Dabdub and R. Manohar. Performance and Portability of an Air Quality Model. *Parallel Computing*, 1997. Special issue on regional weather models.

[DV90] P. Duhamel and M. Vetterli. Fast Fourier Transforms: A Tutorial Review and a State of the Art. *Signal Processing*, 19:259–299, April 1990.

[ERL98] H. El-Rewini and T. G. Lewis. *Distributed and Parallel Computing*. Manning, 1998.

[Fah96a] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, 1996.

[Fah96b] T. Fahringer. On Estimating the Useful Work Distribution of Parallel Programs Under P^3T: Static Performance Estimator. *Concurrency: Practice and Experience*, 8(4):261–282, 1996.

[FC95] I. T. Foster and K. M. Chandy. FORTRAN M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors*, volume 1. Prentice Hall, 1988.

[Fos95] I. T. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. *Proceedings of the Tenth Annual Symposium on Parallel Algorithms and Architectures*, pages 114–118, 1978.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GHR94] E. Gallopoluous, E. N. Houstis, and J. R. Rice. Problem Solving Environments. *IEEE Computer Science and Engineering*, 1(1):11–23, 1994.

[GV94] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[Har96] R. Harley. *Split-Radix FFT Algorithms.* Personal communication, 1996.

[KR90] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Elsevier Science Publications*, 1990.

[Mas96] B. L. Massingill. *The Mesh Archetype.* Technical Report CS-TR-96-25, Computer Science Department, California Institute of Technology, 1996.

[Mas98] B. L. Massingill. *A Structured Approach to Parallel Programming.* Technical Report CS-TR-98-04, Computer Science Department, California Institute of Technology, 1998. PhD thesis.

[MC96] B. L. Massingill and K. M. Chandy. *Parallel Program Archetypes.* Technical Report CS-TR-96-28, Computer Science Department, California Institute of Technology, 1996.

[Mes94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[MKR95] W. Mao, R. Kincaid, and A. Rifkin. *The Impact of Emerging Technologies on Computer Science and Operations Research*, chapter On-Line Algorithms for a Single Machine Scheduling Problem, pages 157–173. Kluwer Academic Publishers, 1995. Stephen Nash and Ariela Sofer, Editors.

[PFTV86] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

[PY79] C.H. Papadimitriou and M. Yannakakis. SIAM Journal of Computing. *SIAM Journal of Computing*, 8:405–409, 1979.

[Rif93] A. Rifkin. Parallel Archetypes. In P.B. Gibbons, R.M. Karp, C.E. Leiserson, and G.M. Papadopoulos, editors, *DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation*, pages 286–293, September 1993.

[RM96] A. Rifkin and B. L. Massingill. *Performance Analysis for Mesh and Mesh-Spectral Archetype Applications*. Technical Report CS-TR-96-27, Computer Science Department, California Institute of Technology, 1996. published in the proceedings of the 1998 International Conference on Parallel and Distributed Processing Technique and Applications (PDPTA'98).

[RRDC97] R. Ramamoorthi, A. Rifkin, B. Dimitrov, and K. M. Chandy. A General Resource Reservation Framework for Scientific Computing. In Y. Ishikawa, R. R. Oldehoeft, J. V. W. Reynders, and M. Tholburn, editors, *Volume 1343 of Springer-Verlag's Lecture Notes in Computer Science*, pages 283–290, December 1997.

[Sch96] E. M. Schooler. *A Multicast User Directory Service for Synchronous Rendezvous*. Technical Report CS-TR-96-18, Computer Science Department, California Institute of Technology, 1996.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[Val90] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[VdV94] E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.

[Win78] S. Winograd. On Computing the Discrete Fourier Transform. *Mathematics of Computation*, 32:175–199, January 1978.

[ZLE91] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.